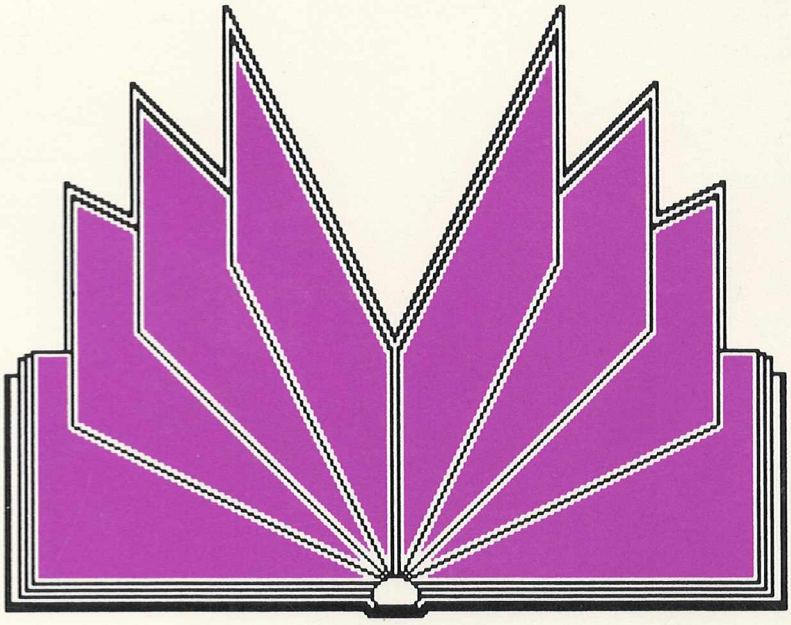
Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER



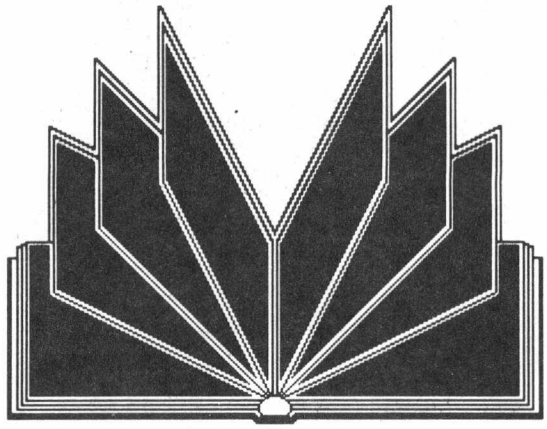
VOLUME

M&T Books

Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER



VOLUME

M&T Books

Acknowledgments

We have always used this space to thank our authors, programmers all, who contribute their time, knowledge, and source code for the edification of other programmers. They deserve all our gratitude: to broadcast hard-earned knowledge like seeds on the wind is extraordinary generosity. We of *Dr. Dobb's* staff are pleased to be the breeze in the metaphor.

Bound Volume Staff

Editor:
Marlin Ouverson

Book Production:
Geoffrey Brandenburg

Index:
Julie Anton

Cover Design:
Shelley Rae Doeden

Publisher's Representative:
Jane Sharninghouse

COPYRIGHT © 1986 by M&T Publishing, Inc. All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Printed in the United States of America

About M&T Publishing, Inc. and Dr. Dobb's Journal

M&T Publishing, Inc. was founded in 1981 as the American subsidiary of Markt & Technik A.G., the leading West German publisher of computer magazines, books and software. In addition to its active role in the international microcomputer marketplace, M&T also publishes two monthly magazines. *Business Software* was acquired in 1983, and in 1984 M&T acquired the rights to publish *Dr. Dobb's Journal* from People's Computer Company.

Since its first issue in 1976, *Dr. Dobb's Journal* has maintained a unique voice among microcomputer publications. The first issue was published to provide short-term distribution of the newly written Tiny BASIC language. Reader response was so strong that it went immediately into monthly production. *DDJ* has led the way since then by focusing on important advances in microcomputers, printing public-domain software and fostering vital reader interaction on technical matters.

Today, *Dr. Dobb's Journal* remains the prime source of information and software tools for advanced programmers. Its early issues are still in demand, and no part of the edtorial content of *DDJ* has ever gone out of print. Volumes One through Nine of this series contain the entire editorial content of *DDJ*'s first nine, years, and are available in bound volumes through M&T Publishing, Inc.

Through *Dr. Dobb's Journal* and its other software publications, M&T Publishing, Inc. is providing microcomputer professionals and enthusiasts with the most advanced and useful software information available.

Editor's Preface

In a short story called "Tlon, Uqbar, Orbis Tertius," Jorge Luis Borges describes the discovery of a strange book. Written in an arcane language, the book seems to be one volume of an encyclopedia of another world, intriguingly unlike the world of everyday reality. The world of the volume rapidly becomes a universal obsession: scholarly journals were devoted to it, people begin to dress and act in ways suggested by the volume. So compelling are the gimpses of the world revealed by the volume that its reality finally crowds out our own, and the world becomes the world of Tlon.

The volume you are holding in your hands is the volume Borges had in mind.

At least it could be. In 1975 a window was opened into another world with the release of the Altair personal computer. A year after, in January 1976, the scholars studying this alien world of personal computers—the programmers—established a journal: Dr. Dobb's Journal of Tiny Basic Calisthenics and Orthodontia. In the following decade high tech became the fashion. Computer technology and computer culture came in through our dashboards and our toaster ovens. Now, bank tellers have turned into hardware and money has become bits transferred over a line. That alien world is ours.

Over the years *Dr. Dobb's Journal of Software Tools* (as the name has evolved) has dedicated itself to supplying the tools that programmers need. Because the computer itself is useless without software, but the software empowers the machine and its user. The computer revolution is truly a software revolution, and *Dr. Dobb's* is a software munitions plant.

The volumes in this series of *Dr. Dobb's Bound Volumes* each contain the complete editorial material for one full year of the jounal—a year of history, a year of valuable programmer's tools. All the articles, all the source code, all the opinions and letters to the editor.

Volume 9 contains programs in many languages, from assembly language to Turbo Pascal to Expert-2, via C, Modual-2, Prolog, and Forth. It includes such powerful tools as two encryption systems, a library for C, and an expert system for weather prediction. But for all the variety, the articles, columns and programs all present powerful tools for professional programmers who are helping to shape the technological world we now inhabit.

Michael Swaine, Editor-in-chief
Dr. Dobb's Journal of Software Tools

Contents

VOLUME 9, NUMBER 87, 1984

3 Editorial
4 Letters
10 Dr. Dobb's Clinic D.E. CORTESI
14 NBASIC: A Structured Preprocessor for MBASIC NAMIR SHAMMAS
21 A Simple Window Package EDWARD MITCHELL
29 Forth to PC-DOS Interface DAVID CORNELL
41 Sorted Diskette Directory Listing for the IBM PC DAN DAETWYLER
54 Emulate WordStar on TOPS-20 STEVEN FISHER
56 CP/M Exchange ROBERT BLUM
60 16-Bit Software Toolbox RAY DUNCAN
65 Book Reviews
67 Software Reviews:
"SuperScripsit" MARLIN OUVERSON
"COMPAC" CHUCK CRAYNE
"MCDISPLAY" MICHAEL FAVITTA
"SMAL/80" DR. GREG LOUIS
78 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 88, 1984

83 Editorial
84 Letters
86 Dr. Dobb's Clinic D.E. CORTESI
89 CP/M Exchange ROBERT BLUM
93 Micro to Mainframe Connection KEITH COYE AND ALVIN GROSSMAN
96 Communications Protocols: Theory and Practice LESLIE BROOKS
99 Unix-to-Unix Network Utilities RON COLEMAN
101 Virtual Personal Computer JIM FLEMING
120 PABX and the Personal Computer TED ROHLING
123 BASIC Language Telecommunications Programming R.S. BROUGHTON
127 U.S. Robotics' S-100 Modem MICHAEL MCKILLIP
130 Software Reviews:
"Friday!" R.A. LANGEVIN
"SAL/80" JIM KRONMAN
"FILEBASE" DENNIS CASHTON
"Oubliette" W. JAMES WIGGINS
134 Book Reviews
136 16-Bit Software Toolbox RAY DUNCAN
143 C/Unix Programmer's Notebook ANTHONY SKJELLUM
146 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 89, 1984

151 Editorial
152 Letters
153 Dr. Dobb's Clinic D.E. CORTESI
155 CP/M Exchange ROBERT BLUM
157 RSA: A Public Key Cryptography System, Part I C.E. BURTON
176 Introduction to PL/C: Programming Language for Compilers MORRIS DOVEY
195 Program Design Using Pseudocode KEN TAKARA
202 More on Binary Magic Numbers DALE WILSON
206 Software Reviews:
"SXR PLUS" CHARLES PETZOLD
"Perfect Writer Perfect Speller" CHARLES K. BALLINGER
208 Book Reviews
211 16-Bit Software Toolbox RAY DUNCAN
215 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 90, 1984

219 Editorial
220 Letters
222 Dr. Dobb's Clinic D.E. CORTESI
223 CP/M Exchange ROBERT BLUM
226 Optimizing Strings in C EDWARD MCDERMOTT
231 Expert Systems and the Weather JACK PARK
236 RSA: A Public Key Cryptography System, Part II C.E. BURTON
253 BASICFMT for TRS-80 (A BASIC Formatter) DAVY CROCKETT
267 Software Reviews:
"microSUB:MATH" ELBERT F. HINSON
"METAFILE Application Development System" MORTON F. KAPLON
275 16-Bit Software Toolbox RAY DUNCAN
282 C/Unix Programmer's Notebook ANTHONY SKJELLUM
284 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 91, 1984

291 Editorial
292 Letters
294 Dr. Dobb's Clinic D.E. CORTESI
296 CP/M Exchange ROBERT BLUM
300 An Introduction to Modula 2 for Pascal Programmers HUGH MCLARTY AND DAVID W. SMITH
305 Converting Fig-Forth to Forth-83 RAY DUNCAN
312 Sixth Generation Computers RICHARD GRIGONIS
322 A New Library for Small-C (Part I) JAMES HENDRIX AND ERNEST PAYNE
341 The Accent Finder EDDY VASILE
353 Solutions to Quirks in dBASE II GENE HEAD
355 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 92, 1984

360 Editorial
361 Letters
362 Dr. Dobb's Clinic D.E. CORTESI
364 CP/M on the Commodore 64 WALT PIOTROWSKI
375 dBASE II Programming Techniques GENE HEAD
379 First Chinese Forth: A Double-Headed Approach TIMOTHY HUANG
386 cc — A Driver for a Small-C Programming System AXEL SCHREINER
397 A New Library for Small-C (Part II) JAMES HENDRIX AND ERNEST PAYNE
406 Comments on "Sixth Generation Computers" MICHAEL DOHERTY
409 Software Reviews:
"Turbo Pascal, v. 1.01" DAVID D. CLARK
413 Book Reviews
414 16-Bit Software Toolbox RAY DUNCAN
417 C/Unix Programmer's Notebook ANTHONY SKJELLUM
433 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 93, 1984

438 Editorial
439 Letters
441 Dr. Dobb's Clinic D.E. CORTESI
444 CP/M Exchange ROBERT BLUM
455 Resident System Extensions: RSX Under CP/M Plus GARRY M. SILVEY
464 p — A Small C Preprocessor AXEL T. SCHREINER
491 A Simple Minimax Algorithm STEVEN A. RUZINSKY
506 Languages and Parentheses: A Suggestion for Forth-like Languages ERNEST E. BERGMAN
512 Balancing Act: The Ultimate Checkbook Balancing Program JOHN E. STITH
513 Book Reviews
516 16-Bit Software Toolbox RAY DUNCAN
521 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 94, 1984

528 Editorial
529 Letters
540 Dr. Dobb's Clinic D.E. CORTESI
543 SCISTAR: Greek and Math Symbols with WordStar JOHN THOMAS AND GUY FLETCHER
547 What's the Diff? A File Comparator for CP/M Plus D.E. CORTESI
558 An Infinite Key Encryption System JOHN A. THOMAS AND JOAN THERSITES
593 A Small-C Concordance Generator JOHN STANEFF
611 16-Bit Software Toolbox RAY DUNCAN
618 C/Unix Programmer's Notebook ANTHONY SKJELLUM
623 Software Reviews:
"PC-Write" RONALD G. PARSONS
"PC Small-C 2.0" RONALD G. PARSONS
625 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 95, 1984

630 Editorial
631 Letters
634 Dr. Dobb's Clinic D.E. CORTESI
639 CP/M Exchange ROBERT BLUM
644 File Maintenance in Forth RAY CADMUS
652 Forth and the Fast Fourier Transform JOE BARNHART
663 Computing with Streams L.L. ODETTE
674 A Forth Native-Code Cross Compiler for the MC68000 RAYMOND BUVEL
704 The FVG Standard Floating-Point Extension RAY DUNCAN AND MARTIN TRACY
708 C/Unix Programmer's Notebook ANTHONY SKJELLUM
712 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 96, 1984

718 Editorial
719 Letters
722 Dr. Dobb's Clinic D.E. CORTESI
725 CP/M Exchange ROBERT BLUM
727 The Software Designer MICHAEL SWAINE
731 More dBASE II Programming Techniques GENE HEAD
734 Simple Calculations with Complex Numbers DAVID D. CLARK
749 GREP.C, A Unix-like Regular Expression Parser in C ALLEN HOLUB
777 16-Bit Software Toolbox RAY DUNCAN
783 C/Unix Programmer's Notebook ANTHONY SKJELLUM
802 Book Reviews
807 Of Interest MICHAEL WIESENBERG

VOLUME 9, NUMBER 97, 1984

812 Editorial
813 Letters
816 Dr. Dobb's Clinic D.E. CORTESI
818 CP/M Exchange ROBERT BLUM
820 Adding Primitive I/O Functions to muLISP MICHAEL CARTER
826 Program Monitor Package ALAN BOMBERGER
843 CP/M 2.2 Goes PUBlic BRIDGER MITCHELL AND DEREK MCKAY
862 A Guide to Resources for the C Programmer TERRY A. WARD
873 RESORT DONALD G. KRANTZ
879 Software Reviews:
"Windows for C, v. 2.00" IAN ASHDOWN
"MYCHESS" RONALD G. PARSONS
882 16-Bit Software Toolbox RAY DUNCAN
894 The Software Designer MICHAEL SWAINE
895 Computer Calisthenics MICHAEL WIESENBERG
896 Book Reviews
902 Of Interest R.P. SUTHERLAND

VOLUME 9, NUMBER 98, 1984

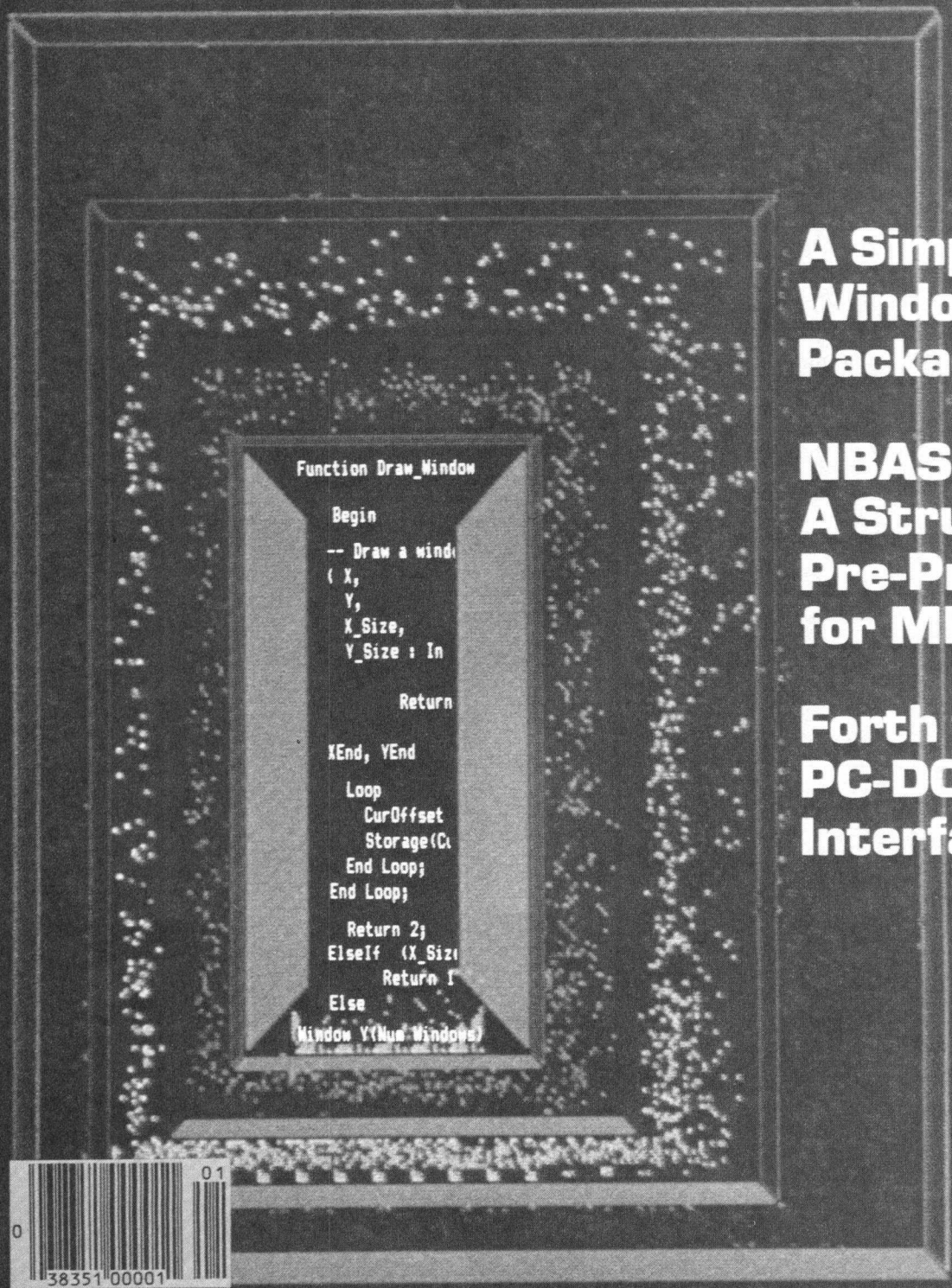
908 Editorial
909 Letters
914 Dr. Dobb's Clinic D.E. CORTESI
916 CP/M Exchange ROBERT BLUM
918 Varieties of Unix — An Introductory Guide to Microcomputer Unixes ALAN WALWORTH
930 Unix Device Drivers JOHN L. BASS
939 A Unix Internals Bibliography JOHN ROGERS
946 A File Browser Program JOHN R. JOHNSON
956 An Introduction to Parsing DR. HENRY A. SEYMOUR
963 16-Bit Software Toolbox RAY DUNCAN
967 C/Unix Programmer's Notebook ANTHONY SKJELLUM
971 Software Reviews:
"NCI COHERENT" A. GOMEZ
"Turbo Pascal v. 2.0" KARL R. KACHIGAN
980 Computer Calisthenics MICHAEL WIESENBERG
981 Of Interest R.P. SUTHERLAND

#87 January 1984

$2.95 [$3.50 in Canada]

Dr. Dobb's Journal

For Users of Small Computer Systems



A Simple Window Package

NBASIC: A Structured Pre-Processor for MBASIC

Forth to PC-DOS Interface

0 01 38351 00001

Dr. Dobb's Journal
For Users of Small Computer Systems

January 1984
Volume 9, Issue 1

CONTENTS

ARTICLES

24 NBASIC: A Structured Preprocessor for MBASIC
by Namir Clement Shammas

This preprocessor provides new commands and features for MBASIC and BASICA programs, including alphanumeric labels, as well as REPEAT and CASE structures. This eliminates the need to use line numbers in branching operations, allowing more structured programs to be written.

36 A Simple Window Package
by Edward Mitchell

This program, written in Augusta (the tiny language introduced last year in *DDJ*), shows how to produce simple windows on your computer display. While written to work on the Osborne computer, the author provides tips on how to modify the package for the IBM PC and other computers.

44 Forth to PC-DOS Interface
by David Cornell

In bringing up Forth on his IBM PC, the author needed a way for Forth to access the resources of MSDOS and the PC's machine language routines. The solution was two special vocabularies for transferring parameters to and from registers during function calls.

60 Sorted Diskette Directory for the IBM PC
by Dan Daetwyler

This handy utility for your IBM PC sorts and formats directory data, and then prints the directory to fit inside the diskette envelope. The program runs under PC-DOS versions 1.x or 2.0, and should operate on any normal configuration.

76 Emulate WordStar on TOPS-20
by Steven Fisher

If you, like the author, are faced with working on many different machines, each with its own editor commands, you too may long for a single command set. This article shows how to make a popular video editor for DEC's TOPS-20 system capable of accepting WordStar commands.

DEPARTMENTS

6 Editorial

8 Letters

18 Dr. Dobb's Clinic

More on optimizing compilers and some problems under CP/M Plus with PAUSE, as well as two public domain programs.

78 CP/M Exchange

A bit on directions for the Exchange and the industry, the PIP mystery device contest, and mapping your I/O ports for serial interfacing.

83 16-Bit Software Toolbox

Feedback on Floating Point Benchmarks, More on DEC Professionals, Linking Assembler and Microsoft BASIC, and an improved CLINK utility.

88 Book Reviews

PC DOS: Using the IBM PC Operating System, Soul of CP/M, Third Caltech Conference on Very Large Scale Integration

90 Software Reviews

SuperScripsit, COMPAC, MCDISPLAY, SMAL/80

104 Of Interest

Publisher – Jane Nissen Laidley
Editorial
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Contributing Editors –
Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg
Marketing
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Marketing Coordinator – Ed Gueble
Advertising
Advertising Director – Beatrice Blatteis
Advertising Sales –
Alice Hinton, Walter Andrzejewski
Circulation
Circulation Director – Terri Pond
Circulation Assistant – Billie Greenwood
Production
Art Director/Production Manager – Barbara Ruzgerian
Production Coordinator – Shelley Rae Doeden
Production Assistants –
Alida Hinton, Tom Martin
Typesetter – Paula Fairchild
Cover Design – Shelley Rae Doeden on Via Video System One
Special Thanks – Berkeley School of Computer Graphics, for technical assistance, and Dale Tucker, for image photography

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

EDITORIAL

Should *DDJ* be a referee for its advertisers? We were recently faced with that question when one company that had advertised with us complained about the wording in the ad its competitor was running with us. Essentially, the complaining party disagreed with claims made in the ad of the competitor; the letter requested, among other things, that we not run such ads in the future. Fortunately, since the ad in question was being rewritten anyway, the matter was resolved amicably.

Advertising is a significant source of information, and the quality of that source is, of course, important. Like most magazines, however, we are not equipped to evaluate all the products and claims that appear in our advertisements. Even if we could, we are not at all sure that screening ads on that basis would be wise. In spite of the difficulties involved, the appropriate party to decide among the array of competing claims and products is the consumer. Our media function of informational support seems best served through such things as reviews, rather than through censorship.

Reynold Wiggins

Changing Your Address?

Staple your label here.

To change your address, attach your address label from the cover of the magazine to this coupon and indicate your new address below.

Name

Address

Address Apt. #

City State

Zip

Mail to: DDJ, P.O. Box E, Menlo Park, CA 94025

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94026. Second class postage paid at Menlo Park, California 94026 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94026 · 415/323-3111 **ISSN 0278-6508**

LETTERS

Nondeterminism Revisited

Dear *DDJ*,

I read with interest L. L. Odette's article, "Nondeterministic Control Words in Forth" (September *DDJ*, No. 83). While I am not particularly enamored by his examples, I think his ideas have great merit and deserve further attention. His rather limited examples may be easily extended to more general constructs.

However, lest some of your readers receive too narrow a view of nondeterministic constructs, I wish to point out that the use of random number generators in these cases is not required. While the use of them does fall under the category of nondeterministic constructs, nondeterminism is a much broader concept than randomly choosing control or search paths.

For example, in Mr. Odette's example, the word CHOOSE on screen #30 (page 52) could be any function that picks one of the several possibilities, including "deterministic" ones. For example, CHOOSE might always return 0 thereby causing ONEOF to always try the first of the two paths presented to it. In this case, we search the "tree of possibilities" in left-to-right or pre-order fashion, rather than a random fashion. By using different functions for CHOOSE in these constructs, we see that we may express our deterministic algorithms in the simpler "nondeterministic" format and still retain their deterministic behavior if desired, thereby having the best of both worlds, so to speak.

I am sure that Mr. Odette is aware of all this, but I felt that it was not emphasized enough in his article.

Sincerely,
Kurt W. Luoto
ROLM Corporation
4900 Old Ironsides Drive
Santa Clara, CA 95050

A Next Alternative

Dear Sir:

Joe Barnhart considered (*Dr. Dobbs*, Sept. 1983) an indirect threaded Forth interpreter in his Listing Three, which uses A1 as a base pointer register. By replacing Forth's 16-bit absolute addresses with 16-bit relative addresses, this would allow a 64K Forth to be located at any address. To use the lowest 64K then, I presume A1 would be kept at 0000 8000H. I don't think it matters whether Barnhart uses the low 16 bits of data or an address register for the index. In either case such 16-bit relative addresses are sign-extended before addition in indirect addressing with index and displacement. I noticed fig-Forth increments both IP and WP after using them. An ADDQ $2, D0 would have to be squeezed into Barnhart's NEXT, before the JMP instruction, if WP is to be incremented. NEXT would then be 12 bytes and 40 clocks.

Here is an alternative NEXT code that is shorter (10 bytes) and faster (32 clocks). Both IP and WP are address registers, which deal in absolute addresses, and are incremented on use. The high 16 bits of D7 indicate which 64K Forth is operating in, and this constant high address functions something like a base page pointer.

```
MOVE.W    (IP)+, D7
MOVE.L    D7, WP
MOVE.W    (WP)+, D7
MOVE.L    D7, SCRATCH
JMP       (SCRATCH)
```

I have not actually tried this because I can't. Address pins 16-23 on my homebrew 68000 system are unused, effectively "decoding" the top 32K addresses to 8000-FFFF. However NEXT is implemented, some attention should be given to concomitant changes in other Forth words that get addresses from CFA and LFA fields, etc.

Finally, though I love the 68000, it would seem much closer to the ideal Forth machine if it had a true indirect JMP with autoincrement. NEXT for a 32K system could then be just 4 bytes and 16 clocks.

Sincerely,
Ken Mantei, Professor
Department of Chemistry
California State College
San Bernardino, CA 92407

We'll C You and Raise You . . .

Dear Editor:

On reading "Looking to C" in the Dr. Dobb's Clinic section of November 1983, I was reminded of my first exposure to C some five or six years ago. My first reaction was negative; but since a number of associates with excellent programming credentials *and accomplishments* were enamored with the language, I delved further and found that C is quite a remarkable language. Of its various strong points, the one that impressed me most was that it *is* a viable alternative to writing in assembly language. I am also impressed by its portability, flexibility, and . . .

I can understand D. E. Cortesi's frustrations in attempting to learn a new language. But his lashing out at C and the Aztec C implementation is not what one would expect from a seasoned, professional programmer. Most importantly, his conclusions and observations are way off base. The C language and the Aztec C implementations are in common use. It is not tomorrow's language; it is *today's*. There are more than 5,000 licensed Aztec C users. This places Aztec C as one of the most widely used C compilers in the micro marketplace. Our users include most of the large impressive companies; but more importantly, there is also an impressive assortment of small and creative companies. Aztec C has been used to produce all or part of a large array of microcomputer software. DRI Japan has recently released a personal CP/M which is written almost entirely in Aztec C. All of our C compilers are written in Aztec C. Our Apple DOS version not only includes a full C compiler, but it also includes a Unix-like command processor – all of which fits neatly and compactly into a 64K 6502 Apple.

Mr. Cortesi did at least take the time to write one program in C. It is poorly written and performs sloppily. Mr. Cortesi goes on to say, "A hypothetical assembly program would be a 2KB object file with a throughput near 1000 bytes/second (if challenged we'll write it to prove it)." This is a meager challenge even in C. Tom Fenwick made some changes to Mr. Cortesi's program that produce a C program that is less than 2KB in size and has a throughput of 1000 bytes/second (see listing on page 12). It is written entirely in Aztec C. The performance increase was achieved by calling the main entry point "Croot," using pointers instead of array references, and using register variables to optimize references to frequently used variables.

The last two techniques are fairly standard and well-known to C programmers. The use of "Croot" is somewhat tricky. The "Croot" routine performs initialization for I/O redirection. Calling the main entry, Croot precludes the unwanted routine from being included by

the linkage editor. The recommended and straightforward way to achieve the same end is to provide a null Croot routine or to remove the reference to Croot from the CALLCPM routine. This technique has been employed by programmers using Aztec C to produce ROMABLE code, code for non-CP/M systems, or for producing small compact programs for the past two years. The only other significant change was to use BDOS calls instead of library routines. This approach is more appropriate in meeting the objectives. We have met Mr. Cortesi's assembly language challenge not in assembly but in Aztec C. A still faster and smaller program was written by replacing Mr. Cortesi's design with one better optimized for size and speed. A few nuances here and there would also make the routine more portable. We'll leave that as an exercise for Mr. Cortesi. Throughout his article Mr. Cortesi uses words like "blunder," "bloated," and "slow." In reading his words all I can see are the frustrations, mistakes, false starts, and errors that are often encountered in informally learning a language by the "read-some, write-some" approach. On our end, better documentation may have reduced some of the frustrations and errors. Consequently, we are currently rewriting our manuals.

Time does not permit responding to all of Mr. Cortesi's remarks. We are now finishing up version 1.06 of Aztec C for its release this month, and version 2.0 for release by 2/84. There is also Z – a C editor based on Vi – due by 11/83. Not to mention the debugger, optimizer, graphics routines, grep, make, the Commodore 64 compiler, the new releases of the IBM PC cross compilers, the new student version, and . . .

If what Mr. Cortesi really meant to say was simply that Aztec C could be improved, we couldn't agree more. We're working on it all the time.

Harry Suckow, President
Manx Software Systems
P. O. Box 55
Shrewsbury, NJ 07701

(Listing begins on page 12)

Dave Cortesi Replies . . .

Aw, come on Mr. Suckow, lighten up. I did not lash out at C; in fact, I ended up saying it was "usable, even elegant," and also "great fun." Nor did I say that Aztec C was bad. On the contrary, I began by noting its good reputation among users in my area as being reliable and standard, and never denied those important virtues later.

My point was that *all* micro compilers (Aztec C included) are simple code generators, unable to produce efficient object code from a straightforward source file. Mr. Fenwick's rewrite of my program is an excellent demonstration. It is much faster and smaller, but look at his methods: in order to make a small, fast program he had to eliminate I/O redirection, avoid the use of standard files (even for error messages!), discard the entire standard C library (even writing his own "strcmp"!), and make it 100% CP/M-dependent.

In short, he had to discard most of the features that make C easy to learn and make programs readable, maintainable, and portable. Surely you see the contradiction between that on one hand and C as a flexible, portable language on the other? I maintain that an intelligent compiler, given my source file, would produce essentially Fenwick's object module, and I maintain that's within the state of the art.

– Dave Cortesi

Personal Interaction in the Fifth Generation Workplace

Dear Editors:

Richard Grigonis writes with confidence and probable accuracy when he describes the hardware and software components of his imaginary fifth generation workstation. His discussions of switching physics and the fallacies of Prolog are clear and sound. But did you really mean to publish that nonsense about the working environment of the future? In a magazine of *Dr. Dobb's* caliber?

When Mr. Grigonis says that our workplaces and jobs will be "a total dedication to efficiency and productivity," that we will work in a "dust-free environment," and that "many employees will be . . . telecommuting," is he serious? Sociologists have shown that one of the reasons that we like to work in offices – and the reason that open-space office design is so popular – is because the office is, first and foremost, a social environment. We don't go to the office just to get work done, as Mr. Grigonis would have it, or surely we would already have been replaced by automatons. We go to the office to communicate with people, to see and talk to others, and only in doing so do we accomplish our tasks. Such social interaction would be quite difficult if we were all enclosed in our fifth generation cocoons, wearing starched whites and respirators to keep from contaminating the sterile air.

The trend today is that people prefer to work in offices, not at home. Electronic mail systems are a feeble substitute for the rich interpersonal contact that the office provides. This is documented by John Naisbitt in his 1983 report, *Megatrends*. Readers of *Dr. Dobb's* will find interesting information in Chapters 1 and 5, which document what our offices are becoming. Naisbitt relies on analysis of what has happened in the past and what is happening in the present to make his judgments, rather than pulling rhetoric out of thin air, as Mr. Grigonis has done.

"Everyone will be an acquaintance of everyone else, but few personal friendships will exist"? Society will decide that "a generalized liberal arts education is too expensive"? "Educational interaction centers"? People transformed into "users"? Our children brought up to serve machines? Come on, Mr. Grigonis. Get serious, Dr. Dobb's. Man is made up of more than just rationality; people are warm, communicating individuals, with a need to be with others. Get the picture?

Yours cordially,
Christopher Palermo
546 Cecelia Court
Los Altos, CA 94022

BASIC Dates

Dear Doc,

I have a few comments about Gordon King's "Julian Dates for Microcomputers" (June 1983 issue, p. 66). I only recently saw this article. My apologies if this reaches *DDJ* after other letters on the same subject.

The term Julian date usually refers only to the kind of dates calculated by astronomers. What Mr. King is actually referring to is a days-between-dates algorithm (of which there are many).

Mr. King is obviously not a BASIC programmer! His comment that 16-bit arithmetic is not enough for making the conversions between Julian date and calendar date is basically (pun intended!) correct. He is of the opinion that most BASICs can only handle such limited arithmetic and, therefore, that they cannot be used to translate his pseudo higher-level language code into a BASIC program. This is quite wrong!

It ends up that many BASICs can handle higher-bit arithmetic (less than 16-bit is quite rare). As my example, I have enclosed a listing for his pseudo code (I call his Julian dates "pseudo" too just to keep the flavor!). The listing (page 15) is written in North Star BASIC which does not have a MODulus function. I tried running the examples shown using 8-, 10-, and 14-digit BASICs. 8-digit only fails when near (and above) 65536. The calendar dates come out only one day short. 10-digit gives correct calendar dates for Julian dates over 65536. The 14-digit BASIC calculates in both directions with the most accuracy and was used for the printed example.

I have no idea if the calculations for

(Continued on page 105)

Revised C Program (Text begins on page 8)

```
#define MAXIDS  500
#define MAXLEN  16

struct shortfcb {
        char drive;
        char name[8], type[3];
} ;

struct fcb {
        char drive;
        char name[8], type[3];
        char ext, s1, s2, rc;
        char ptrs[16];
        char crec;
} outfcb;

#define inpfcb  ((struct fcb *) 0x5c)
#define FCB2    ((struct shortfcb *) 0x6c)
#define BUFFER  ((char *) 0x80)
#define BUFFEND ((char *) 0x100)
#define EOFCHAR 0x1a

char *lp = BUFFEND;

/* use structure here to allow easy copy of data by assignment */
struct line {
        char text[MAXLEN];
} ;


struct line *table, *last;
struct line *settop();

Croot()
{
        register char *cp;

        *(struct shortfcb *)&outfcb = *FCB2;    /* save second file parameter */

        if (bdos(15, inpfcb) == 0xff)
                quit("Cannot open input file$");

        table = settop(0);
        do {
                if ((last = settop(sizeof(struct line))) == (struct line *)0)
                        quit("Too many symbols$");
        } while (readline(last->text));

        quick(table,last-1);

        bdos(19, &outfcb);
        if (bdos(22, &outfcb) == 0xff)
                quit("Cannot make output file$");
        lp = BUFFER;

        while (table < last) {
                cp = (table++)->text;
                while (*cp)
                        outbyte(*cp++);
                outbyte('\r');
```

```
                outbyte('\n');
        }

        while (lp > BUFFER)
                outbyte(EOFCHAR);
        bdos(16, &outfcb);
        bdos(0);
}

readline(buff)
char *buff;
{
        register int i = 0;

        for(;;) {
                if (lp >= BUFFEND) {
                        if (bdos(20, inpfcb) != 0)
                                break;
                        lp = BUFFER;
                }
                switch (buff[i] = *lp++) {
                case EOFCHAR:
                case '\n':
                        goto done;
                case '\r':
                        continue;
                }
                if (i < MAXLEN-1)
                        ++i;
        }
done:
        buff[i] = 0;
        return i;
}

outbyte(c)
{
        *lp++ = c;
        if (lp >= BUFFEND) {
                if (bdos(21, &outfcb) != 0)
                        quit("Out of disk space!$");
                lp = BUFFER;
        }
}

swap(p,q)
register struct line *p, *q;
{
        struct line t;

        t = *p; *p = *q; *q = t;
}

strcmp(s1,s2)
register char *s1, *s2;
{
        int c;

        while (*s1)
                if ((c = *s1++ - *s2++) != 0)
```

(Continued on next page)

Revised C Program (Listing continued, text begins on page 8)

```
                        return c;
        return 0;
}

quick(lo, hi)
struct line *lo, *hi;
{
        register struct line *i, *j;

        if (lo < hi) {
                for (i=lo, j=hi; i < j;) {
                        while (i < j && strcmp(i->text,hi->text) <= 0)
                                ++i;
                        while (j > i && strcmp(j->text,hi->text) >= 0)
                                --j;
                        if (i < j)
                                swap(i, j);
                }
                swap(i, hi);
                quick(lo, i-1);
                quick(i+1, hi);
        }
}

quit(cp)
char *cp;
{
        bdos(9, cp);
        bdos(0);
}
```

End Listing

BASIC Dates (Text begins on page 10)

```
+GO BASIC14
North Star Basic on Horizon, Version 5.5.0
14 digit precision
READY
LOAD PJDATE
READY
LIST

10 REM WRITTEN BY GORDON KING IN PSEUDO HIGHER-LEVEL LANGUAGE
20 REM FROM DR. DOBB'S JOURNAL, JUNE, 1983, NO. 80, P.66
30 REM ENTERED AS A NORTH STAR BASIC PROGRAM BY SAUL G. LEVY, TUCSON, ARIZONA,
40 REM SEPTEMBER 22, 1983
50 REM
60 REM THE ZERO DATE IS MARCH 1, 1900
70 ! CHR$(26),"PSEUDO JULIAN DATE (FROM MARCH 1, 1900 = 0)"
80 !
90 INPUT "GIVE:  1)DATE  2)JD  3)END: ",Z
100 IF Z=1 THEN 160
110 IF Z=2 THEN 270
120 IF Z=3 THEN END
130 GOTO 90
140 REM
150 REM CALENDAR TO PSEUDO JULIAN DATE
160 INPUT "MONTH,DAY,YEAR: ",M,D,Y
170 Y1=Y-1900
180 IF M<=2 THEN 210
190 M1=M-3
200 GOTO 220
210 M1=M+9 \ Y1=Y1-1
220 J=INT(1461*Y1/4)+INT((153*M1+2)/5)+D-1
```

```
230 ! "PSEUDO JULIAN DATE IS",J
PRESS RETURN TO CONTINUE
240 GOTO 80
250 REM
260 REM PSEUDO JULIAN TO CALENDAR DATE
270 INPUT "PSEUDO JULIAN DATE: ",J
280 Y1=INT((4*J+3)/1461)
290 D2=(4*J+3)/1461 \ D3=INT(D2) \ D4=D2-D3 \ D1=INT(D4*1461+.001)
300 Y=Y1+1900
310 D1=INT(D1/4+1)
320 M1=INT((5*D1-3)/153+.001)
330 D2=(5*D1-3)/153 \ D3=INT(D2) \ D4=D2-D3 \ D1=INT(D4*153+.001)
340 D=INT(D1/5+1)
350 IF M1>=10 THEN 380
360 M=M1+3
370 GOTO 390
380 M=M1-9 \ Y=Y+1
390 ! "DATE IS",M,D,Y
400 GOTO 80
410 REM LAST LINE
READY
RUN
```

End Listing

Sample Run of BASIC Dates

```
PSEUDO JULIAN DATE (FROM MARCH 1, 1900 = 0)

     GIVE:  1)DATE  2)JD  3)END: 1
     MONTH,DAY,YEAR: 3,1,1900
     PSEUDO JULIAN DATE IS 0

     GIVE:  1)DATE  2)JD  3)END: 2
     PSEUDO JULIAN DATE: 0
     DATE IS 3 1 1900

     GIVE:  1)DATE  2)JD  3)END: 1
     MONTH,DAY,YEAR: 12,31,1977
     PSEUDO JULIAN DATE IS 28429

     GIVE:  1)DATE  2)JD  3)END: 2
     PSEUDO JULIAN DATE: 28429
     DATE IS 12 31 1977

     GIVE:  1)DATE  2)JD  3)END: 1
     MONTH,DAY,YEAR: 1,1,2000
     PSEUDO JULIAN DATE IS 36465

     GIVE:  1)DATE  2)JD  3)END: 2
     PSEUDO JULIAN DATE: 36465
     DATE IS 1 1 2000

     GIVE:  1)DATE  2)JD  3)END: 1
     MONTH,DAY,YEAR: 8,5,2079
     PSEUDO JULIAN DATE IS 65536

     GIVE:  1)DATE  2)JD  3)END: 2
     PSEUDO JULIAN DATE: 65536
     DATE IS 8 5 2079

     GIVE:  1)DATE  2)JD  3)END: 1
     MONTH,DAY,YEAR: 1,1,2100
     PSEUDO JULIAN DATE IS 72990

     GIVE:  1)DATE  2)JD  3)END: 2
     PSEUDO JULIAN DATE: 72990
     DATE IS 1 1 2100

     GIVE:  1)DATE  2)JD  3)END: 1
     MONTH,DAY,YEAR: 1,1,3000
     PSEUDO JULIAN DATE IS 401715

     GIVE:  1)DATE  2)JD  3)END: 2
     PSEUDO JULIAN DATE: 401715
     DATE IS 1 1 3000

     GIVE:  1)DATE  2)JD  3)END: 1
     MONTH,DAY,YEAR: 1,1,100000
     PSEUDO JULIAN DATE IS 35830965

     GIVE:  1)DATE  2)JD  3)END: 2
     PSEUDO JULIAN DATE: 35830965
     DATE IS 1 1 100000

     GIVE:  1)DATE  2)JD  3)END: 3
     READY
     BYE
     +OD #0
```

End Listing

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

Optimizing Compilers, Cont.

Remember our November grumblings about the lack of optimizing compilers on micros? Maybe you didn't believe what we said about global optimization being a well-known technology, or doubted that such technology could be applied to micros. Allow us to refer you to a paper in the September *Communications of the ACM,* entitled "A Practical Tool Kit for Making Portable Compilers."

Its authors (A. S. Tannenbaum, H. van Stavern, E.G. Keizer, and J.W. Stevenson), working at Vrije University in Amsterdam, report on a package of software that eases the creation of optimizing compilers for a variety of languages with a variety of target computers.

Tannenbaum, et al., have implemented the old idea of UNCOL, the "universal computer oriented language." The concept is one in which there is a separate "front end" for each language, where a "front end" is a simple compiler producing unoptimized output in the form of instructions for a simple virtual machine – rather like UCSD p-code.

There is a "back end" for each target machine, a program that translates the UNCOL code into machine language for one real CPU – rather like the p-code-to-Z80 assemblers that are available now.

Between the front and back ends one places machine-independent programs whose job is to optimize the p-code produced by any front end.

I repeat, the paper reports not a plan or design but an implementation. The impressive part of it is the attention paid to optimization. Bear in mind that a "front end" is equivalent to the UCSD p-system compiler and to many one-pass compilers for Pascal and C that are sold as finished products.

The first "middle-end" program is a peephole optimizer, similar to (but rather more thorough than) that incorporated in any micro compiler we know of, with the possible exception of Digital Research's PL/I. Besides spotting verbose p-code sequences and emitting more efficient ones, it:

- notes constant expressions and reduces them to single constants,
- applies "strength reduction," replacing expensive operations like multiplication with cheap ones like shift,
- eliminates null operations like offsetting 0 bytes to the first item of a record,
- consolidates consecutive moves to adjacent locations into block moves,
- eliminates unreachable code, and
- eliminates branches-to-branches.

We are already beyond the abilities of most compilers, and nowhere near finished. The output of this first pass goes to the global optimizer. It examines the entire program and creates a data-flow graph. Based on the graph, it can decide to eliminate a small procedure by expanding its code in-line at the points at which it is called. It recognizes loops and searches them for invariant expressions that can be moved outside to one-pass code. It collects information on the lifespan of variables. When two variables of similar size are never "live" at the same point, it can make them occupy the same storage. It recognizes sequential scans over arrays and can replace subscript computation with a simple pointer-increment operation. In some cases it reorders expressions into "forms that are likely to produce better object code," as when it spots A+B*C – likely to require two temporary registers – and converts it into B*C+A – likely to need only a single accumulator.

These are all standard optimizations, but current micro compilers do few or none of them. The optimized output, which is still in the form of p-code, is the input to a "back end" program, which translates it into the assembly language of some target machine. That's still not the end! Tannenbaum, et al., have found that it is still useful to put the assembly source through one more pass of peephole optimization, this time looking for machine-dependent sequences that could be tightened up.

Now, *this* is what we mean by an optimizing compiler – and these guys have made a tool kit for its creation. When this level of compiler technology reaches the micro market, we may finally be able to let go of assembly language for all but the most critical modules. It can't happen too soon!

The Ambiguous Open

David McLanahan of New Jersey is one of our correspondents who is experimenting with CP/M 3, and he has found two pieces of public domain software that glitch under it. One is version 5 of the CRC program, which does a Cyclic-Redundancy Check (CRC) of the files on a disk. "The default execution of CRC (vers. 5)," McLanahan writes, "compares its output with any one of a class of CRC-containing files that it tries to find on the disk. The problem is that it calls for 'CRCLIST.???' where the queries will correspond to a disk number.

"CP/M 3 will not accept the wildcards in the BDOS call, and the program bombs. I fixed it by changing the call to CRCLIST.CRC, but this reduces the generality of the file search."

We know what's going on there, and it's a pitfall that more people should know about. In CP/M 2, a program could open a file giving an ambiguous filespec. The first matching file in the directory would be opened. That feature was later banned in MP/M 2 and subsequent versions of the BDOS, we suppose for security reasons. A program has to present an explicit filespec to open now. The directory-search services can be used to find the first, or any, file matching an ambiguous name.

Two-Bit Problem

McLanahan's second public-domain "gotcha" is in a program called SD (for Super Directory). Its version 4.3 "works fine under CP/M 3 with the exception of the free-space calculation which comes out high. I have no idea of a fix."

We suspect that SD reports too much free space because it isn't taking into account the two-bit allocation vector in CP/M 3. Previous BDOSs logged in every drive after a warm start. Accordingly, their allocation vectors (bit-maps which reflect the state of all free blocks on a disk) were refreshed at the end of every program's run. That isn't so in CP/M 3 (or MP/M-86, or Concurrent CP/M). The latest BDOSs don't log in a disk until they have to.

That leaves the possibility that a program will terminate without closing an output file. Space allocated to that file will not be recorded in the directory, but it would show as in use in the old, one-bit-per-block allocation vector, and so would be unusable until the disk was logged in again. For a hard disk, that might not be until the next power-up.

To prevent this, the new BDOSs use two bits for each block on a drive. They set one when a block is allocated to a file, but set the second only when the directory entry that records this block is written to disk. At warm start time, they sweep over the allocation vector and free any blocks that are marked to be allocated in only one of the two bits.

Programs that compute the amount of free space on a drive by counting 0 bits in the allocation vector will now, naturally, give the wrong numbers. For these systems, such programs should give up counting bits and use the much simpler method of asking the BDOS itself, using the new "Get Free Space" service, BDOS function 46.

Incidentally, McLanahan has reported both problems to SIG/M, purveyor of these programs.

The Pause That Didn't

Some time back, while we were promoting the use of CP/M submit files, we published a PAUSE command. Its function is to stop a submit file and wait for one key from the operator. If that key is Control-C, PAUSE cancels the submit file: if not, it lets execution continue.

Well, our old PAUSE won't work in CP/M 3. It pauses all right, but it can't cancel the batch job. It tries to do that by erasing the file $$$.SUB, but in CP/M 3, *there isn't one*. The much-improved submit function works off of a file named SYSIN*nn*.$$$ instead, where *nn* is an unpredictable pair of digits. Incidentally, it starts at 58 in our system and 47 in Bob Blum's, and goes down by one for every level of nesting of submit files. PAUSE can't erase a file whose name it can't predict.

Even if it could, erasing the SYSIN file won't stop batch execution. Unlike previous systems, which put one command in each 128-byte record of $$$.SUB, CP/M 3 packs command lines at their natural lengths in the SYSIN file; and it reads them 128 bytes at a time into a buffer in a Resident System Extension (RSX) which resides at an unpredictable location in storage. At the moment PAUSE is executed, several following command lines are already in storage, and they will be executed whether it erases the SYSIN file or not.

A lot of our submit files depend on PAUSE, and as we came to rely more and more on CP/M 3 it got to be a real problem that we couldn't use them. So we called up Bob Blum. "Hey, Bob," we said, "you're the CP/M columnist around here; give us a PAUSE command, will you?"

"OK, son," he said. "Ahem. You stand right there for 30 seconds, you hear?" He waited. "How was that?"

A Brief Digression

There is nowhere near enough humor in computer mags these days, we think; so the little we find is all the nicer. In that regard, we'd like to recommend "The Gamer's Cafe," a column by Rodney Gambicus in *80 Micro*. Gambicus has created a surreal soap opera based on a couple of computer-crazed loons roving the country in a van stuffed with TRS-80s. He does a pretty fair pastiche of Hunter S. Thompson, and some of his gags are hilarious. Recently, the van drifted to the home of one of the characters to view the new baby brother.

> "A wonderful baby," Mercedes cuddled him. "And he's got five little toes on his foot! How many toes, baby? For T equals 1 to 5: Count T: Next!"
>
> Well, maybe ya hadda been there....

But We Digress

Since Blum couldn't pause in his work to help us, we had to do it ourselves. Therefore in the listing (below) you will find the source of a PAUSE command that adapts itself according to the system it runs in. In CP/M 2 it cancels the submit file in the old way. In CP/M 3 it has to talk to the RSX that implements SUBMIT.

An RSX is a piece of code that is loaded as part of a transient program. The CCP recognizes a special header on the COM file and relocates the RSX code to high storage, just below the BDOS. The BDOS jump vector is retargeted so that all BDOS calls go to the RSX, which is supposed to pass on those it doesn't handle. A whole chain of RSXs can be installed, each handling some services and passing the others on.

The SUBMIT command prepares a SYSIN file containing the substituted lines of batch commands. It primes its RSX (which loaded as part of SUBMIT) with the name of the file, and ends. The RSX then intercepts all calls for terminal input and satisfies some or all of them from the SYSIN text. Somehow it distinguishes between the CCP asking for a command line and other programs asking for input. When the file is used up, it turns itself off, and its storage is reclaimed at the following warm start.

In order to cancel the submit file, then, our PAUSE has to communicate with the RSX and somehow persuade it to end itself early. We disassembled the RSX and managed to make enough sense out of its rather convoluted code to find out how this could be done.

There is a BDOS service number, 60, which has been reserved for direct communication with RSXs. The argument to it is the address of a byte in storage where an RSX function number has been placed. Each RSX in the chain is supposed to check the number and deal with it if appropriate. If the call gets passed all the way up to the BDOS, it returns an error code.

It turns out that the submit RSX will respond to RSX function code 129 by erasing its SYSIN file and disabling itself. Always, we think, but maybe only sometimes. So the way of killing a submitted job in CP/M 3 is to call for BDOS service 60, pointing to a byte of 129.

There's one more complication. In CP/M 2, a nested submit file (made possible by SUBMIT mods published here) is simply stacked on the end of the one file, $$$.SUB. If submit files were nested, PAUSE would cancel all of them at once. In CP/M 3, a nested submit file is a separate SYSIN file read by a nested RSX. Here PAUSE can only cancel the innermost one – the file that invoked it. The outer jobs will continue to run.

The code in the listing will work as well for cancelling a submit from a program like QUITIF. We leave upgrading the QUITIF command, and figuring out how to make SKIPIF work in CP/M 3, to readers. Send solutions to Bob Blum, please; next month we are gonna muscle in on Ray Duncan's turf instead.

DDJ

(Listing begins below)

Dr. Dobb's Clinic (Text begins on page 18)

```
;============================================================
;
;       PAUSE  <any sort of a message>  (CP/M 2 or 3)
;
; Used in a submit file, the PAUSE command sends a BEL to the
```

(Continued on page 22)

```
; console and reads one byte from the console.  If the byte
; received is NOT a Control-C, batch execution continues.
; When Control-C is typed, the active submit file is cancelled.
;
; Use PAUSE as a way of halting a batch job when operator
; action is needed, or when the operator must make a decision
; as to whether to continue or cancel.  The command tail
; may be used to convey (or complete) a message, as in
;
;       ; mount other disk in the B-drive and hit RETURN
;       pause ...or hit control-c to cancel job
;
;==========  Cancelling a Job in CP/M 2.2
;
; The submit file is cancelled by erasing A:$$$.SUB.  This
; cancels the batch file at once, even if submits were
; nested.  The presence of XSUB has no effect on PAUSE.
;
;==========  Cancelling a Job in CP/M 3.0 (CP/M PLUS)
;
; The method of cancelling a submit file is not documented,
; and was derived by studying a disassembly of the very
; confusing Resident System eXtension (RSX) which implements
; batch execution.  Being undocumented, it ain't guaranteed.
;
; If the batch file containing the PAUSE line is nested,
; only the active file is cancelled.  Execution resumes in
; the outer file, the one that called the active one.
;
; If program input is being taken from the submit file,
; (i.e. if the next line in the file starts with "<"), then
; the byte read by PAUSE will come from the file.  This is
; probably not a useful feature.
;=============================================================
Bdos    equ     5       ; good old bdos
OutByte equ     02      ; write-a-byte
Toot    equ     07      ; ascii beep value
InByte  equ     01      ; read-a-byte BDOS function
CtlC    equ     03      ; control-C
WhatSys equ     12      ; get-system-id
CpmPlus equ     31h     ; id of CP/M Plus (BDOS 3.)
;
        mvi     e,Toot
        mvi     c,OutByte
        call    Bdos    ; "squeep," saith you, sirrah?
        mvi     c,InByte
        call    Bdos
        cpi     CtlC    ; cancel or continue?
        rnz             ; continue... back to work
        mvi     c,WhatSys
        call    Bdos    ; we cancel, but how?
        cpi     CpmPlus
        jnc     KillPlus ; very sophisticatedly...
```

```
;
; CP/M 2.2 -- kill by erasing $$$.SUB (which has to be
; on the A-drive or it wouldn't be executing).
;
EraFile equ     19      ; erase a file
;
        lxi     d,SubFCB
        mvi     c,EraFile
        call    Bdos    ; A=FFh if no such file
        jmp     Gloat
;
SubFCB  db      01,'$$$     SUB',0,0,0,0
;
; CP/M 3.0 or above -- kill by calling the RSX which is feeding
; file lines into the system, passing it the sub-argument
; KillGet.  If there is no RSX active, the call will reach
; the BDOS instead, and it will return A=FFh.
;
CallRSX equ     60      ; Bdos number reserved to RSX calls
KillGet equ     81h     ; code tells GET RSX to end itself
RSXparm db      KillGet ; DE points here for RSX
;
KillPlus:
        lxi     d,RSXparm ; DE->argument for RSX
        mvi     c,CallRSX ; code says, "hey, RSXs, look"
        call    Bdos    ; A=FFh if no RSX responded
;
; We have terminated the active batch job, or maybe there
; was none.  If there was one, type a message.
;
TypeMsg equ     09      ; type-a-string
;
Gloat:
        inr     a       ; was there a RSX or file?
        rz              ; (no, A=FF)
        lxi     d,Message ; yes, claim responsibility
        mvi     c,TypeMsg ; ..for the bombing
        call    Bdos
        ret
;
Message db      'Submit file cancelled.',13,10,'$'
        end
```

End Listing

NBASIC: A Structured Preprocessor for MBASIC

NBASIC, a preprocessor for MBASIC and BASICA programs, offers new commands and features to programmers. These features are:

- The use of alphanumeric labels in branching
- The REPEAT-UNTIL loop
- The CASE-OF structure for multiway IF statements
- Fortran-like subroutine calls

These features mainly enable the programmer to avoid the use of line numbers in branching operations, which means programs can now be written in a more structured way. The role of NBASIC is to process these programs and convert them into syntax accepted by the BASIC interpreter.

NBASIC does not have extensive error checking; it keeps the amount of code to a minimum in order to maximize the number of lines it can process.

"The role of NBASIC is to process these programs and convert them into syntax accepted by the BASIC interpreter."

NBASIC Labels

Four old keywords may be used with NBASIC labels. They are:

- GOTO[*label*]
- GOSUB[*label*]
- THEN[*label*]
- ELSE[*label*]

A label is defined with the form:

[LBL:*label*]

All spaces are significant within square brackets. As an example, if NBASIC is given this program:

```
10  [LBL:START] INPUT"ENTER X ";X
20  IF X<=0 THEN[QUIT]
30  PRINT"X = ";X
40  X = X - 1
50  IF X>0 THEN[START]
60  [LBL:QUIT] END
```

it will produce this output:

```
10  INPUT"ENTER X ";X
20  IF X<=0 THEN 60
30  PRINT"X = ";X
40  X = X - 1
50  IF X>0 THEN 10
60  END
```

Labels may be used more than once on the same line, as in:

```
IF X>0 THEN[MOVE] ELSE[QUIT]
```

A label may be defined anywhere on a line, as long as the definition is not the last item on that line. A label definition, however, should not appear alone on a line; at least one remark should follow it:

```
100 [LBL:HERE] ' remark appended
```

The REPEAT-UNTIL Loop

The REPEAT-UNTIL structure allows a program to go through a block of commands at least once and then to test whether a repeat is needed. The general usage is:

```
REPEAT
    statements
UNTIL ( test )
```

These loops may be nested. The variables I1, I2, and so on are reserved for loop control and should not be used within the REPEAT-UNTIL loop. Here is a sample:

```
10  INPUT"ENTER X ";X
20  REPEAT
30  PRINT"X = ";X
40  X = X - 1
50  UNTIL (X >= 0)
60  END
```

From that program, NBASIC will produce:

```
10  INPUT"ENTER X ";X
20  FOR I1=0 TO 1 : ' REPEAT
30  PRINT"X = ";X
40  X = X - 1
50  I1=-1*(X >=0):
    NEXT I1 : ' UNTIL (X >=0)
60  END
```

The REPEAT keyword may be preceded by an NBASIC label. Otherwise, both the REPEAT and UNTIL keywords should be alone on their lines; nothing should precede or follow them.

The CASE-OF Structure

The CASE-OF structure allows multiple multiline IF statements. Its general form is:

```
CASE OF ( expression 0 )
!( expression 1 ) DO
    statement group 1
!( expression 2 ) DO
    statement group 2
!ELSE DO
    default statements
ENDCASE
```

The CASE-OF command tests the value of *expression 0* against *expression 1, expression 2,* and so forth, and executes only the group of statements that corresponds to a successful test. The !ELSE DO line is offered as a "when all else fails do . . ." option. Here is an example:

```
10  INPUT"ENTER X ";X
20  CASE OF (X)
30  !(0) DO
40  PRINT"X IS 0"
50  !(1) DO
60  PRINT"X IS 1"
70  !ELSE DO
80  PRINT"SOMETHING ELSE"
90  ENDCASE
100 END
```

From that, NBASIC will produce:

```
10  INPUT"ENTER X ";X
20  I0 = (X)
30  IF I0 <> (0) THEN 50
40  PRINT "X IS 0"
50  IF I0 <> (1) THEN 70
60  PRINT"X IS 1"
70  DUMY=1 ' DUMMY STATEMENT
80  PRINT"SOMETHING ELSE"
90  DUMY=1 ' DUMMY STATEMENT
100 END
```

A CASE-OF structure may have up to fifty statement blocks. NBASIC labels may be used anywhere in the blocks, but the structures may not be nested. No more than one !ELSE DO line should appear in any CASE.

by Namir Shammas

Namir Clement Shammas, 1533F Honey Grove Drive, Richmond, Virginia 23229.

Copyright © 1983 Namir Clement Shammas. All rights reserved.

Permission is granted for non-commercial use and distribution. Commerical use without the author's written permission is forbidden.

Subroutine Calls

Although the NBASIC subroutine feature simulates Fortran calls, the subroutines are actually part of the main program and may not have local variables.

An NBASIC subroutine begins with a SUB statement in this form:

SUB "*name*" TAKES *input-list*
GIVES *result-list*

The keyword SUB introduces the line and the *name* in quotes identifies the subroutine for use in a CALL statement. The TAKES and GIVES clauses are optional; they define the interface to the subroutine. The *input-list* is a list of variable names through which the subroutine receives its input. The *result-list* is a list of variable names that will contain the results when the subroutine ends. In both lists, names are separated by semicolons. Here are some examples of subroutine declarations:

- SUB "TEST" TAKES A;B;C GIVES E
- SUB "MATRIX" TAKES A(1,I) GIVES B(J)
- SUB "TRY"
- SUB "ROOT" GIVES FOFX

A subroutine ends with a RETURN statement.

The CALL statement is used to call a subroutine. The TAKES and GIVES clauses in the CALL statement are matched up with the same clauses in the called subroutine. The variable names are matched by position; they do not have to be exactly the same in the CALL and SUB statements. NBASIC generates assignment statements to copy the caller's TAKES variables into the subroutine's, and to copy the subroutine's GIVES variables into the caller's. For example, the program:

```
10 INPUT"ENTER X ";X
20 CALL "TEST" TAKES X GIVES X;Z
30 PRINT Y
40 PRINT Z
50 END
60 SUB "TEST" TAKES A
       GIVES B(1);B(2)
70 B(1) = A*A
80 B(2) = SQR(A)
90 RETURN
```

will be converted by NBASIC into:

```
10 INPUT"ENTER X ";X
20 A=X: GOSUB 60: Y=B(1): Z=B(2)
30 PRINT Y
40 PRINT Z
50 END
60  'SUB "TEST" TAKES A
       GIVES B(1);B(2)
70 B(1) = A*A
80 B(2) = SQR(A)
90 RETURN
```

Subroutine calls may be nested, as long as variable names do not conflict. Recursion, however, is not allowed. The CALL and SUB statements should be alone on their lines; nothing should precede or follow them. A blank must come before and after the keywords TAKES and GIVES.

Using NBASIC

NBASIC is a BASIC program. To use it, first edit the program to be converted and save it as an ASCII file (i.e., SAVE "*filename*", A). Then load NBASIC and RUN it. If NBASIC detects an error, it will halt with a message. If no errors are detected, the converted program will be saved and its filename displayed. The converted program is also in ASCII form; you can load it and either edit or run it. You may want to adjust its indentation or change the generated code.

How the Program Works

The NBASIC program listing (page 27) functions in the following way.

Line 1040 sets the dimensions (which may be altered, memory permitting) for the number of lines and other parameters.

Lines 1100-1190 obtain the program filename and read the file to the string array L$(I) while counting the number of lines read. The LINE INPUT command ensures that lines containing commas are not interpreted as several strings.

The loop in lines 1210-1240 converts any REM into the apostrophe that serves the same purpose; it is easier to handle only one remark symbol.

The REPEAT-UNTIL feature is handled in lines 1260 to 1470. NBASIC searches each program line for the tokens REPEAT and UNTIL. The variable CNT, which serves as a record keeper, is incremented by one for every REPEAT token found and decremented by one for every UNTIL token. CNT monitors the proper sequencing and matching of the two tokens and allows nested loops. The REPEAT is replaced by a FOR statement and its corresponding NEXT statement replaces the UNTIL token. The test expression that was originally with the UNTIL token is incorporated into a statement affecting the value of the loop counter prior to its NEXT statement. If the test fails, the loop counter is set to zero so as to repeat the loop. Otherwise, it is set to one to exit the loop.

Lines 1490 to 1790 convert the alphanumeric labels into the actual line numbers they are associated with. In the loop lying within lines 1500 to 1570, the program scans the entire user program for alphanumeric labels, storing whatever labels are found in the string array LBL$ and the corresponding line numbers in LSTK, while removing the labels from the line. If the label is alone in a line, the line will be reduced to nothing, which is why an alpha label line should contain additional text, at least a remark! The loop in lines 1590 to 1690 scans for the NBASIC token words used with alphanumeric labels, making sure that the alpha labels mentioned in these tokens are valid.

The CASE-OF feature is handled in lines 1950 to 2440. Here NBASIC scans for the token word CASE OF and expects to find at least one !(expression) and an ENDCASE token. The number of expressions, their locations, and the presence and location of the !ELSE DO token are monitored; and the target of the GOTO inserted at the end of each case option block is determined. The !ELSE DO and ENDCASE tokens are transformed into dummy statements, while the !(expression) sequences are turned into IF statements. NBASIC deals with the CASE-OF feature on a "case by case" basis, thus no nesting is allowed.

Lines 2630 to 3220 tackle the CALL feature. Again, NBASIC scans the entire user program searching for the SUB token. When one is found, the program studies the line to extract the subroutine name and the input/output variable lists. For the latter lists, the program tests their presence, counts the number of variables in each list, and learns their names. The syntax of a list relies only on the semicolon, so as to avoid confusion when both arrays and expressions are used in the lists. The names of the variables in the input/output lists are stored in the matrix string NM$. To keep memory requirements at a minimum once a subroutine is declared, NBASIC scans the entire user program to convert any calls to that subroutine.

In the loop residing in lines 2960 to 3210, the program starts to scan the user program for subroutine CALLs. Here the program studies the CALL statement to match it with the subroutine name that is declared. The input/output variable lists are examined (if available) and a replacement line, stored in L$, is formed; it contains the line number where the subroutine call was originated and the GOSUB statement. The GOSUB is preceded by assignments to the subroutine input list variables and followed by assignments to the output list variables named in the call. It is worth mentioning that, since the input/output variable lists are similar, NBASIC surrounds them with braces and brackets. This avoids code repetition and makes scanning the lists more systematic.

The rest of the program stores, displays, and prints the converted program.

■■J

(Listing begins on next page)

NBASIC Listing (Text begins on page 24)

```
1000 '      BASIC PREPROCESSOR  VERSION  1.1   July 6, 1983
1010 '      COPYRIGHT (c) 1983  by     N. C. SHAMMAS
1020 OPTION BASE 1
1030 DEFINT A-Z
1040 DIM L$(5000),LSTK(50),LBL$(50),NM$(2,30),VAR(2),B$(2),C$(2)
1050 CLS ' CLEAR SCREEN FOR THE IBM PC
1060 PRINT : PRINT : PRINT
1070 DATA "BASIC  PREPROCESSOR  V1.1",
          "COPYRIGHT (c) 1983 by N. C. SHAMMAS"
1080 FOR I=1 TO 2 : READ L$(I) : NEXT I
1090 FOR I=1 TO 2 : PRINT SPC(40-LEN(L$(I))/2);L$(I) : PRINT : NEXT I
1100 INPUT "ENTER FILENAME ";F$ : PRINT : PRINT
1110 PTR = INSTR(F$,".") : IF PTR=0 THEN 1130
1120 FOUT$=MID$(F$,1,PTR)+"BAZ"
1130 OPEN "I",1,F$
1140 L = 0
1150 WHILE EOF(1)<>-1
1160     L = L + 1
1170     LINE INPUT #1,L$(L)
1180 WEND
1190 CLOSE #1
1200 N=L
1210 FOR I=1 TO N
1220     P=INSTR(L$(I),"REM")
1230     IF P <> 0 THEN L$(I)=MID$(L$(I),1,P-1)+"'"+MID$(L$(I),P+3)
1240 NEXT I
1250 '  REPEAT . . . UNTIL( test )
1260 CNT = 0
1270 FOR I=1 TO N
1280     PTR = INSTR(L$(I),"REPEAT")
1290     IF PTR=0 THEN 1360
1300     P=INSTR(L$(I),"'")
1310     IF (P<PTR) AND (P>0) THEN 1360
1320     CNT = CNT +1
1330     A$=STR$(CNT)
1340     I$="I"+RIGHT$(A$,LEN(A$)-1)
1350     L$(I)=MID$(L$(I),1,PTR-1)+"FOR "+I$+"=0 TO 1"+" :' "+
                                                    MID$(L$(I),PTR)
1360     PTR = INSTR(L$(I),"UNTIL")
1370     IF PTR=0 THEN 1460
1380     P=INSTR(L$(I),"'")
1390     IF (P<PTR) AND (P>0) THEN 1460
1400     IF CNT <=0 THEN PRINT "UNTIL ERROR IN LINE ";VAL(L$(I)) : END
1410     A$=STR$(CNT)
1420     I$="I"+RIGHT$(A$,LEN(A$)-1)
1430     A$=L$(I)
1440     L$(I) = MID$(A$,1,PTR-1)+" "+I$+"=-1*"+MID$(A$,PTR+5)+" : NEXT "+
              I$+ ":' "+MID$(A$,PTR)
1450     CNT = CNT - 1
1460 NEXT I
1470 IF CNT <> 0 THEN PRINT "REPEAT ERROR : MORE REPEATs THAN UNTILs" :
                  END
1480 '   LABEL CONVERSIONS
1490 STK = 0
```

```
150Ø FOR I= 1 TO N
151Ø     PTR=INSTR(L$(I),"[LBL")
152Ø     IF PTR=Ø THEN 157Ø
153Ø     A$=L$(I) : STK=STK + 1 : PTR2=INSTR(A$,"]")
154Ø     LSTK(STK)=VAL(A$)
155Ø     LBL$(STK)= MID$(A$,PTR+5,PTR2-PTR-5)
156Ø     L$(I)=MID$(A$,1,PTR-1)+MID$(A$,PTR2+1)
157Ø NEXT I
158Ø IF STK=Ø THEN 194Ø
159Ø FOR I=1 TO N
16ØØ     A$=L$(I)
161Ø     PTR=INSTR(A$,"THEN[")
162Ø     IF PTR <> Ø THEN 171Ø
163Ø     PTR=INSTR(A$,"ELSE[")
164Ø     IF PTR <> Ø THEN 171Ø
165Ø     PTR=INSTR(A$,"GOTO[")
166Ø     IF PTR <> Ø THEN  171Ø
167Ø     PTR=INSTR(A$,"GOSUB[")
168Ø     IF PTR <> Ø THEN 17ØØ
169Ø NEXT I : GOTO 194Ø
17ØØ J = 6 : GOTO 172Ø
171Ø J=5
172Ø PTR2=INSTR(A$,"]")
173Ø A$= MID$(A$,PTR+J,PTR2-J-PTR) : FLAG=Ø
174Ø FOR K=1 TO STK
175Ø     IF A$ = LBL$(K) THEN FLAG=K : K=STK
176Ø NEXT K
177Ø IF FLAG=Ø THEN PRINT "ERROR IN LINE ";VAL(L$(I));" LABEL NOT FOUND" :
              END
178Ø L$(I)= MID$(L$(I),1,PTR+J-2)+STR$(LSTK(FLAG))+MID$(L$(I),PTR2+1)
179Ø GOTO 16ØØ
18ØØ '    CASE OF(exp)
181Ø '    !(exp1) DO
182Ø '
183Ø '    statements
184Ø '
185Ø '    !(exp2) DO
186Ø '
187Ø '    statements
188Ø '
189Ø '    !ELSE                    <-- OPTIONAL
19ØØ '
191Ø '    statements
192Ø '
193Ø '    ENDCASE
194Ø DEF FNL$(A$)=RIGHT$(STR$(VAL(A$)),LEN(STR$(VAL(A$)))-1)
195Ø NE = Ø : N1 = Ø : STK = Ø
196Ø FOR I=1 TO N
197Ø    PTR=INSTR(L$(I),"CASE OF")
198Ø    IF PTR=Ø THEN 244Ø
199Ø    A$=L$(I)
2ØØØ    P=INSTR(A$,"'")
2Ø1Ø    IF (P<PTR) AND (P>Ø) THEN 263Ø
2Ø2Ø    PTR=INSTR(A$,"(")
```

(Continued on next page)

NBASIC Listing (Listing continued, text begins on page 24)

```
2030     I$=MID$(A$,PTR)
2040     L$(I)=FNL$(A$)+" IØ ="+I$
2050     STK=Ø : NE=Ø : J=I : N1=Ø
2060     J=J+1
2070     IF J >= N THEN PRINT "ERROR IN 'CASE OF': NO (exp n) FOUND!" :END
2080     PTR=INSTR(L$(J),"!(")
2090     IF PTR=Ø THEN 2140
2100     STK = STK + 1
2110     LSTK(STK) = J
2120     N1 = 1
2130     GOTO 2060
2140     PTR=INSTR(L$(J),"!ELSE")
2150     IF PTR=Ø THEN 2190
2160     LE = J
2170     NE = NE + 1
2180     GOTO 2060
2190     PTR=INSTR(L$(J),"ENDCASE")
2200     IF PTR=Ø THEN 2060
2210     IF N1=Ø THEN PRINT "ERROR " : END
2220     IF NE > 1 THEN  PRINT "ERROR :MORE THAN ONE '!ELSE' PER 'CASE OF'":
          END
2230     IF STK=Ø THEN 2630
2240     L$=": GOTO"+STR$(VAL(L$(J)))
2250     L$(J)=FNL$(L$(J))+" DUMY=1 ' DUMMY STATEMENT "
2260     L = LSTK(STK)
2270     A$ = L$(L)
2280     PTR=INSTR(A$,"!(")+1
2290     PTR2=INSTR(A$,"DO")
2300     IF NE=Ø THEN  LE=J : GOTO 2330
2310     L$(LE) = FNL$(L$(LE))+" DUMY=1 ' DUMMY STATEMENT"
2320     L$(LE-1)=L$(LE-1)+L$
2330     L$(L)=FNL$(A$)+" IF IØ <> "+MID$(A$,PTR,PTR2-PTR)+" THEN"+
                STR$(VAL(L$(LE)))
2340     IF STK=1 THEN 2630
2350     FOR J=1 TO STK-1
2360              L = LSTK(J)
2370              M = LSTK(J+1)
2380              A$= L$(L)
2390              PTR=INSTR(A$,"!(")+1
2400              PTR2=INSTR(A$,"DO")
2410              L$(L)=FNL$(A$)+" IF IØ <> "+MID$(A$,PTR,PTR2-PTR)+"THEN"+
                        STR$(VAL(L$(M)))
2420              L$(M-1) = L$(M-1) + L$
2430     NEXT J
2440 NEXT I
2450 '
2460 ' 1505  CALL "MATRIX"  TAKES A;B;D GIVES E;F
2470 '
2480 '             statements
2490 '
2500 ' 9000 SUB "MATRIX" TAKES X;Y;Z GIVES L;M
2510 '
2520 '             subroutine body
2530 '
```

```
2540 '   RETURN
2550 '
2560 '
2570 '   THE CALL ON LINE 1505 WILL BECOME
2580 '
2590 '   1505     X=A : Y=B : Z=C : GOSUB 9000 : E=L : F=M
2600 '
2610 '
2620 '
2630 B$(1)="(" : B$(2)="[" :C$(1)=")" : C$(2)="]"
2640 FOR I=1 TO N
2650     P=INSTR(L$(I),"GOSUB")
2660     IF P <> 0 THEN 3220
2670     P=INSTR(L$(I),"SUB")
2680     P1=INSTR(L$(I),"'")
2690     IF (P=0) OR ((P1<P) AND (P1>0)) THEN 3220
2700     A$=L$(I)
2710     SBLN = VAL(A$)
2720     P1=INSTR(A$,CHR$(34))
2730     P2=INSTR(P1+1,A$,CHR$(34))
2740     IF (P1*P2)=0 THEN PRINT "ERROR IN SUBROUTINE NAME @ LINE ";SBLN :
         END
2750     SBNM$=MID$(A$,P1+1,P2-P1-1)
2760     L$(I)=FNL$(A$)+" '"+MID$(A$,LEN(FNL$(A$))+1)
2770     VAR(1) = 0 : VAR(2) = 0
2780     GOSUB 3400 : ' INSERT BRACES AND BRACKETS
2790     FOR J=1 TO 2
2800              P=INSTR(A$,B$(J))
2810              IF P=0 THEN 2950
2820              P1=P
2830              P2=INSTR(A$,C$(J))
2840              P=INSTR(P1+1,A$,";")
2850              PTR=INSTR(A$,"[") : IF PTR=0 THEN PTR=LEN(A$)
2860              IF (P=0) OR ((P>PTR)  AND (J=1)) THEN 2920
2870              VAR(J)=VAR(J)+1
2880              K=VAR(J)
2890              NM$(J,K)=MID$(A$,P1+1,P-P1-1)
2900              P1 = P
2910              GOTO 2840
2920              VAR(J)=VAR(J) + 1
2930              K = VAR(J)
2940              NM$(J,K) = MID$(A$,P1+1,P2-P1-1)
2950     NEXT J
2960     FOR J=1 TO N
2970              P = INSTR(L$(J),"CALL")
2980              IF P=0 THEN 3210
2990              A$ = L$(J)
3000              P1=INSTR(A$,CHR$(34))
3010              P2=INSTR(P1+1,A$,CHR$(34))
3020              IF P1*P2=0 THEN 3210
3030              IF (SBNM$ <> MID$(A$,P1+1,P2-P1-1)) THEN 3210
3040              GOSUB 3400 : ' INSERT BARCES AND BRACKETS AS MARKERS
3050              L$ = FNL$(A$)
3060              FOR L= 1 TO 2
```

(Continued on page 34)

NBASIC Listing (Listing continued, text begins on page 24)

```
3070                    IF VAR(L)=0 THEN 3180
3080                    P1=INSTR(A$,B$(L))
3090                    P2=INSTR(A$,C$(L))
3100                    FOR K=0 TO VAR(L)
3110                            IF K=0 THEN K=K+1
3120                      IF K=VAR(L) THEN P=P2
                                        ELSE P=INSTR(P1+1,A$,";")
3130                      IF P=0 THEN PRINT "ERROR:SEMICOLON MISSING IN";
                                                " LINE ";VAL(A$) : END
3140                      IF L=1 THEN
                                  L$=L$+" "+NM$(L,K)+"="+
                                        MID$(A$,P1+1,P-P1-1)+":"
3150                      IF L=2 THEN
                                  L$=L$+" "+MID$(A$,P1+1,P-P1-1)+"="+
                                        NM$(L,K)+":"
3160                      P1 = P
3170                NEXT K
3180                IF L=1 THEN L$=L$+" GOSUB"+STR$(SBLN)+":"
3190            NEXT L
3200            L$(J)=L$
3210      NEXT J
3220 NEXT I
3230 OPEN "O",1,FOUT$
3240 PRINT "CONVERSION IS SUCCESSFUL !"
3250 PRINT : PRINT "PREPARE PRINTER AND PRESS ANY KEY " : PRINT
3260 A$=INPUT$(1) : CLS
3270 FOR I = 1 TO N
3280      A$=L$(I)
3290      PRINT A$
3300      LPRINT A$
3310      PRINT #1,A$
3320 NEXT I
3330 CLOSE#1
3340 FOR I=1 TO 79 : PRINT "_"; : NEXT I : PRINT
3350 PRINT : PRINT "THE ABOVE LISTING IS RESIDING IN FILE '";FOUT$;"'"
3360 END
3370 '
3380 '
3390 ' SUBROUTINE TO INSERT BRACES AND BRACKETS
3400      P1 = INSTR(A$,"TAKES") : P2 = INSTR(A$,"GIVES")
3410      IF P1 = 0 THEN 3440
3420      MID$(A$,P1+5,1)="{"
3430      IF P2 = 0 THEN A$ = A$+"}" ELSE MID$(A$,P2-1,1)="}"
3440      IF P2 = 0 THEN 3470
3450      MID$(A$,P2+5,1) = "["
3460      A$ = A$ + "]"
3470      RETURN
```

End Listing

A Simple Window Package

by Edward Mitchell

Windows are used to split a computer screen into several possibly overlapping viewing areas. Like a tiny screen, each window may be used to show a separate program or task (see Figure 1, page 37).

Windows are used extensively in the Xerox Star, VisiCorp's new VisiON operating system for the IBM PC, and are the basis of the Apple Lisa and the soon to be announced MacIntosh computers. This article shows how you can create and use windows on your computer. While this article is geared towards the Osborne user, tips are included for modifying the routines for the IBM PC and other computers.

"Windows are like sheets of paper; rather than piling sheets of paper on top of one another, our software simply draws windows on top of whatever else is on the screen."

Why Windows?

To understand the reason for windows, imagine a desk covered with sheets of paper. When we stop working on one project, we don't have to take everything off the desk. Instead, we can just plop the new work on top of the old.

Windows are like those sheets of paper: rather than piling sheets of paper on top of one another, our software simply draws windows on top of whatever else is on the screen. The new project or task is completed within the window boundaries, the window then disappears, and the original text is restored – just like working on a desk top.

Figure 1 shows a typical computer in use. At (a) the word processor is in operation. At (b) we momentarily leave the word processor to check our electronic mailbox, so a window is created for the MAIL program. Windows can also be placed on top of windows, as shown in (c), where a quick check of the file directory is made.

The Window Package

Five procedures create and manipulate the windows. The window package is shown in Listing One (page 38), written in Augusta, a subset language based on Ada (see references 1 through 4).

DRAW_WINDOW draws a new window on the screen, saving the text that was originally displayed. DRAW_WINDOW can create windows anywhere on the screen, even on top of other windows. WRITE_TEXT outputs text to an existing window. INSERT_ROW and DELETE_ROW add and remove a single line, respectively, from within a window and can be used for text editing and scrolling. Finally, REMOVE_WINDOW completely erases the most recently drawn window, restoring the original text to the screen.

Some caveats: This package is a simple implementation, lacking many advanced features found in other products such as the Apple Lisa. For example, if one window is displayed on top of the other, you should write only text to the topmost window. Windows can be removed only in the reverse order from that in which they were drawn. And windows cannot be moved from one location to another nor scaled to different sizes. Some features, such as moving data between windows, are easy to implement, but they are not shown here.

Despite these limitations, the package has numerous applications. In fact, very few programs will ever need more advanced features.

Implementation Details

When a window is created, the original text on display is saved in an array called STORAGE, which is indexed with the variable CUROFFSET. The function GETCH(X,Y) returns the character on the screen at cursor location (X,Y). A character is then saved in STORAGE with the statements:

```
STORAGE(CUROFFSET) :=
    GETCH ( X, Y );
CUROFFSET := CUROFFSET + 1;
```

STORAGE is used like a stack. As text is saved, CUROFFSET is incremented. Consequently, the most recently saved text corresponds to the most recently drawn window.

A window is erased by decrementing CUROFFSET, fetching the character from the array, and drawing it on the screen; this is repeated for all characters within the window. Because of this simple scheme for storing text, windows cannot be drawn and erased at random but must be erased in the reverse order from that used to draw them.

For each window, the system remembers the (X, Y) location of the upper left corner of the window (array variables WINDOW_X() and WINDOW_Y()), plus the width and height of the window (variables WINDOW_XSZ() and WINDOW_YSZ()).

Function DRAW_WINDOW checks to see that array STORAGE has enough room to save the screen's current text. If enough memory is available, the coordinates and size of the new window are saved, and the text where the window will be drawn is written to the STORAGE array. The window border, consisting of hyphens "-" on the top and bottom edges and vertical bars "|" on the sides, is then sketched in place, and the window's interior is filled with blanks.

Windows are numbered consecutively from 0 according to the order of creation. The first window drawn on the screen is number 0, the second is number 1, and so on.

Text is written inside a window by calling WRITE_TEXT and passing to it the window number, the X and Y locations relative to the window's upper left corner (e.g., (1,1) is row 1, column 1, within the window), and the string to be written.

REMOVE_WINDOW erases the window that was just drawn; this is equivalent to the sheet of paper on top of all the others on a desk.

Two editing procedures, INSERT_ROW and DELETE_ROW, manipulate the contents of the windows. INSERT_ROW creates a new blank line within the window, sliding all of the lines below it down one line position. DELETE_ROW removes a line, sliding all those that are below it up one line and leaving a blank line at the bottom of the window.

GETCH, shown in Listing One coded

Edward Mitchell, P. O. Box 390145, Mountain View, California 94039. (Please enclose a self-addressed, stamped envelope if you wish a reply.)

for the Osborne 1, reads characters from the screen by accessing the Osborne's memory-mapped video display. Characters are written to the screen simply by changing a *value in* memory. The same character, once it's on the screen, can be read by accessing the memory location in the memory-mapped display area. The PEEK() in GETCH and the POKE in PUTCH access these memory locations. See references 5 and 6 for additional details concerning the Osborne 1 video display system.

Different computer systems require different code for GETCH and PUTCH. Listing Two (page 43) shows versions of GETCH and PUTCH written in IBM Pascal for the IBM Personal Computer. In Listing Two, (a) is a global variable declaration needed to access screen memory, (b) initializes the global variable, and (c) is the code for GETCH and PUTCH.

Summary

Windows are being used increasingly as a means to improve and simplify the user interface. The five procedures just described provide a simple implementation of windows for the Osborne 1 and are easily modified for use on other computers, including the IBM PC. These routines are not nearly as powerful as the windowing capabilities of the Xerox Star or Apple Lisa, but they will suffice for many of your applications.

References

1. Edward Mitchell, "Augusta, An Ada Subset for Micros," *Dr. Dobb's Journal,* January 1983, p. 14.
2. _____, "Augusta, Part II – The Augusta P-Code Interpreter," *Dr. Dobb's Journal,* March 1983, p. 20.
3. _____, "Augusta, Part III – The Augusta Compiler," *Dr. Dobb's Journal,* May 1983, p. 13.
4. _____, "Augusta, Part IV – The Augusta Compiler (Continued)," *Dr. Dobb's Journal,* July 1983, p. 12.
5. *Osborne 1 User's Reference Guide,* Osborne Computer Corporation.
6. *IBM Personal Computer Technical Reference,* International Business Machines Corporation, 1981. See pages 2-37 through 2-63.

Note: The Augusta compiler, including source, p-code interpreter, disassembler, debug utility, and 160-page reference guide, is available for Z80-based CP/M computers for $90 from: Laboratory Microsystems, 4147 Beethoven St., Los Angeles, CA 90066; (212) 306-7412.

DDJ

(Listing begins on page 38)

(a)

```
+-----------------------------------------------------------------------+
|                                                                       |
|    Windows are like those pieces of paper.  But rather than           |
|    piling pieces of paper on top of one another, our software         |
|    simply draws windows on top of whatever else is on the             |
|    screen.  The new project or task is then completed within          |
|    the window boundaries, the window disappears and the               |
|    original text is restored.  Just like how we do our work           |
|    on a desk top.                                                     |
|                                                                       |
|    Figure 1(a), (b), and (c) show a typical computer use:  At         |
|    (a) is a word processor in operation.  At (b) we                   |
|    momentarily leave the word processor to check our                  |
|    electronic mail box.  A window is created on top of the            |
|    existing text.  But windows can even be placed on top of           |
|    other windows, as shown in (c), where a file directory             |
|    listing is shown.                                                  |
|                                                                       |
|                                                                       |
|....+....1....+....2....+....3....+....4....+....5....+....6...       |
+-----------------------------------------------------------------------+
```

(b)

```
+-----------------------------------------------------------------------+
|                                                                       |
|    Windows are like those pieces of paper.  But rather than           |
|    piling pi-----------------------------------, our software         |
|    simply dr|                                  | is on the          |
|    screen.  | MAIL SYSTEM                      |pleted within       |
|    the windo| -----------                      | and the            |
|    original |                                  | do our work        |
|    on a desk| Messages Received:  3            |                    |
|             |                                  |                    |
|    Figure 1(|  From          Date     Time     |puter use:  At      |
|    (a) is a |  ----          ----     ----     |) we                |
|    momentari|  John          1/24/83  0935     |k our               |
|    electroni|  Kim           1/23/83  1547     | top of the         |
|    existing |  Sam           1/23/83  1117     |ed on top of        |
|    other win|                                  | directory          |
|    listing i|                                  |                    |
|             | Command?                         |                    |
|             ------------------------------------                    |
|....+....1....+....2....+....3....+....4....+....5....+....6...       |
+-----------------------------------------------------------------------+
```

(c)

```
+-----------------------------------------------------------------------+
|                                                                       |
|    Windows are lik-------------------------------ather than          |
|    piling pi------|                              |our software        |
|    simply dr|     |  File Directory of B:        |s on the            |
|    screen.  | MAIL|  --------------------        |eted within         |
|    the windo| ----|                              |nd the              |
|    original |     |  WINDOW.TXT                  |o our work          |
|    on a desk| Mess|  ARTICLE                     |                    |
|             |     |  AUGUSTA.COM                 |                    |
|    Figure 1(|  Fr |  KEYWORDS.TXT                |ter use:  At        |
|    (a) is a |  --|  RUN.COM                      |we                  |
|    momentari|  Jo|  ARTICLE.BAK                 |our                 |
|    electroni|  Ki|                              |op of the           |
|    existing |  Sa|                              | on top of          |
|    other win|    |------------------------------|irectory            |
|    listing i|                                   |                    |
|             | Command?                          |                    |
|             -------------------------------------                   |
|....+....1....+....2....+....3....+....4....+....5....+....6...       |
+-----------------------------------------------------------------------+
```

Figure 1.

Windows provide a way of presenting several tasks or programs on a single screen. At (a) is a typical word processing program. (b) shows the same screen but with a window superimposed on top of the word processor, allowing access to an electronic mail utility. And (c) shows a file directory window on top of the mail window.

Simple Window Package (Text begins on page 36)

Listing One

```
Procedure Windows Is

-- Simple Implementation of Windows
-- For the Osborne I Portable Computer.

-- Written in AUGUSTA, A Subset Language
-- Based on Ada.

-- By Edward Mitchell, 18 Jan 1983
-- This program is available for public use
-- and is not copyrighted.



Max_Windows     : Constant := 8;        -- Max # of windows allowed
Max_Storage     : Constant := 4095;     -- Max storage available
ScreenHeight    : Constant := 31;       -- Height and width of screen
ScreenWidth     : Constant := 127;
Num_Windows,                            -- Current # of windows defined
CurOffset       : Integer;              -- Next Free Byte in Storage

Window_X        : Array (Max_Windows) Of Integer;
Window_Y        : Array (Max_Windows) Of Integer;
Window_XSz      : Array (Max_Windows) Of Integer;
Window_YSz      : Array (Max_Windows) Of Integer;
Storage         : Array (Max_Storage) Of Character;

----------------------------------------
Procedure PutCh
       -- Write 'Ch' To screen at (X, Y)
       ( X, Y : Integer; Ch : Character ) Is
Begin -- This will only work on the Osborne 1
  Poke ( -4096 + Y*128+ X, Ch );
End;
----------------------------------------
Function GetCh
       -- Return Char. at screen (X, Y)
       ( X, Y : Integer ) Return Character Is
Begin -- This will only work on the Osborne I
  Return Character ( Peek ( -4096 + Y*128 + X ) );
End;
----------------------------------------
Procedure Draw_Row
       -- Output a row of characters, 'LEN'
       -- bytes long
       (XS,                -- X and Y location to
        YS,                -- begin the row
        Len : Integer;     -- # chars in row
        Ch : Character) Is -- Actual char to print

Begin
  For I In 0..Len - 1
  Loop
    PutCh ( XS + I, YS, Ch );
```

(Continued on next page)

Simple Window Package
Listing One

(*Listing continued, text begins on page 36*)

```
End Loop;
End;
----------------------------------------
Function Draw_Window
        -- Draw a window on the screen
        ( X,                -- (X,Y) position of
          Y,                -- upper left corner
          X_Size,           -- Width and height of
          Y_Size : Integer) -- the window
                            -- Return NON-ZERO If error
              Return Integer Is

       XEnd, YEnd      : Integer;
Begin
  -- Determine if there is enough memory in Storage() to
  -- save the text that is currently in the window area
  If  Num_Windows = Max_Windows  Then
    Return 2;
  ElseIf  (X_Size+2) * (Y_Size+2) > (Max_Storage - CurOffset)  Then
        Return 1;  -- Return 1 meaning out of memory
  Else

    -- Save Window Parameters
    Window_X(Num_Windows) := X;
    Window_Y(Num_Windows) := Y;
    Window_XSz(Num_Windows) := X_Size;
    Window_YSz(Num_Windows) := Y_Size;
    Num_Windows := Num_Windows + 1;

    -- Copy Existing text where window will be placed
    -- And save the text in the Storage() Array
    XEnd := X + X_Size + 1;
    YEnd := Y + Y_Size + 1;
    For TempY In Y..YEnd
    Loop
      For TempX In X..XEnd
      Loop
        CurOffset := CurOffset + 1;
        Storage(CurOffset) := GetCh ( TempX, TempY );
      End Loop;
    End Loop;

    -- Draw Window Border and Clear Text Area
    Draw_Row ( X, Y, X_Size+2, '-');    -- Top Border
    For TempY In Y+1..YEnd-1            -- Clear out interior
    Loop                                -- and draw sides
      Draw_Row ( X, TempY, X_Size+2, ' ');
      PutCh ( X, TempY, '|');
      PutCh ( X+X_Size+1, TempY, '|');
    End Loop;
    Draw_Row (X,YEnd, X_Size+2,'-');    -- Bottom border
    Return 0;
  End If;
```

```
   End;
   ----------------------------------------
   Procedure Remove_Window Is
        -- Erase the most recent window drawn
  Y,
  Y_Size,
  X,
  X_Size        : Integer;
Begin
  Num_Windows := Num_Windows - 1;
    -- Just repaint the original text from
    -- the last window
  Y := Window_Y(Num_Windows);
  Y_Size := Window_YSz(Num_Windows);
  X := Window_X(Num_Windows);
  X_Size := Window_XSz(Num_Windows);

  For TempY In Reverse Y + Y_Size + 1 .. Y
  Loop
    For TempX In Reverse X + X_Size + 1 .. X
    Loop
      PutCh ( TempX, TempY, Storage(CurOffset) );
      CurOffset := CurOffset - 1;
    End Loop;
  End Loop;
End;
----------------------------------------
Procedure Write_Text
       -- Print A String inside a window
       ( Window_Num,       -- # Of Window to write to
         XRel,             -- (X,Y) location, relative
         YRel:Integer;     -- to window to draw string
         S:Out String) Is  -- The string to print

       Len,
       Y,
       X_Size,
       X       : Integer;
Begin
  X := Window_X(Window_Num);
  X_Size := Window_XSz (Window_Num);
  Y := Window_Y(Window_Num);

  -- Check that the string will actually fit in the
  -- window.  If it doesn't, then only print the
  -- string's first bytes that do fit.  Ignore the
  -- remainder.
  If  (S'Length + XRel - 1) <= X_Size  Then
        Len := S'Length;
        Else
               Len := X_Size - XRel + 1;
  End If;

  For I In X+XRel .. X+XRel+Len-1
  Loop
```

(Continued on next page)

Simple Window Package

Listing One

(Listing continued, text begins on page 36)

```
    PutCh(I, Y+YRel, Char (@S,I-(X+XRel)+1));
  End Loop;
End;
-----------------------------------------
Procedure Insert_Row
        -- Insert a row of blanks at 'ROW'
        (Window_Num,      -- # Of Window to operate on
         Row:Integer) Is  -- Window Relative Row #

        Y,
        Y_Size,
        X,
        X_Size  : Integer;
Begin
  Y := Window_Y (Window_Num);
  X := Window_X (Window_Num);
  X_Size := Window_XSz (Window_Num);
  Y_Size := Window_YSz (Window_Num);
        -- Copy each line within the window down
        -- one line position
  For I In Reverse Y+Y_Size .. Y+Row+1
  Loop
    For J In X+1 .. X+X_Size
    Loop
       PutCh(J,I,GetCh(J,I-1));
    End Loop;
  End Loop;

  For I In X+1 .. X+X_Size
  Loop  -- Then, Set the new row to all blanks
    PutCh(I,Y+Row,' ');
  End Loop;
End;
-----------------------------------------
Procedure Delete_Row
        -- Delete one row of text from window
        (Window_Num,      -- # Of window to use
         Row:Integer) Is  -- Window relative row to delete
  X,
  Y,
  X_Size,
  Y_Size        :       Integer;

Begin
  X := Window_X ( Window_Num );
  Y := Window_Y ( Window_Num );
  X_Size := Window_XSz ( Window_Num );
  Y_Size := Window_YSz ( Window_Num );
        -- Copy each row into the previous line
        -- on the screen
  For I In Y+Row .. Y + Y_Size
  Loop
```

```
    For J In X+1 .. X+X_Size
    Loop
      PutCh(J,I,GetCh(J,I+1));
    End Loop;
  End Loop;

  For I In X+1 .. X+X_Size
  Loop
    PutCh(I,Y+Y_Size,' ');
  End Loop;
End;


Procedure Demonstration Is
  Dummy : Integer;
  Ch    : Character;
Begin
  For  I In 1 .. 20
  Loop
    Put_Line("THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG");
  End Loop;

  Dummy := Draw_Window ( 10,3, 30, 15);
  Write_Text (0, 2,2, "Mail System");
  Write_Text (0, 2,3, "-----------");
  Write_Text (0, 2,5, "Messages Received:  3");
  Write_Text (0, 4,7, "From          Date     Time");
  Write_Text (0, 4,8, "----          ----     ----");
  Write_Text (0, 4,9, "John          1/24/83  0935");
  Write_Text (0, 4,10,"Kim           1/23/83  1547");
  Write_Text (0, 4,11,"Sam           1/23/83  1117");

  GetChar(@Ch);
  Remove_Window;

End;


Begin
                      -- Initialize Window System
  Num_Windows := 0;   -- Variables
  CurOffset := 0;
  PutChar(26);        -- Clear Osborne I screen
  Demonstration;

End;
```

End Listing One

(Continued on next page)

Simple Window Package

Listing Two

(Listing continued, text begins on page 36)

```
GETCH and PUTCH routines needed to access screen memory on the IBM
Personal Computer, using IBM Pascal.  See reference 3 and the IBM
Pascal Guide for additional details.  The attribute should be set
to 7 for normal display of text.

(a)

     Var
          ScreenPtr : Ads of Array [0..1] Of Char;


(b)
     (* If Using the monochrome display, then use *)

          ScreenPtr.S := 16#B000;

     (* But if using the color graphics card then use *)

          ScreenPtr.S := 16#B800;


(c)

Function GetCh ( X, Y : INTEGER; Var Attribute : INTEGER) : CHAR;
Begin
                            (* Make ScreenPtr point to (X,Y)
                               position in screen  memory *)
  ScreenPtr.R := WRD ( 2 * X + 160 * Y );
                            (* Read character's attribute value *)
  Attribute := ORD ( ScreenPtr^[1]);
                            (* And finally read the actual character *)
  GetCh := Ord ( ScreenPtr^[0]);
End;


Procedure PutCh ( X, Y, Attribute : INTEGER; CH : CHAR );
Begin
  ScreenPtr.R := WRD ( 2 * X + 160 * Y );
  ScreenPtr^[1] := Chr(Attribute);
  ScreenPtr[0] := Ch;
End;
```

End Listing Two

Forth to PC-DOS Interface

by David Cornell

After receiving my IBM PC, my first project was to install a version of Forth. Using the MS-DOS Assembler with block disk I/O, I "brought up" Fig Forth. The disk I/O was coded as part of the assembler source using routines in the IBM ROM. Forth was loaded from MS-DOS as a COM file; and disk access was the standard 1024-byte block accessing absolute locations on the disk. This provided a very simple Forth system. Most importantly, it got things to the point where, with great relief, EDLIN, .ASM files, .EXE files, .OBJ files, .LST files, .MAP files, CREF, LINK, and EXE2BIN could be abandoned, and the rest of the Forth system was simply written in Forth itself.

Some interface was still required between Forth and the PC's machine language routines. Cursor positioning was needed for a screen editor, and access to the resources of MS-DOS was necessary. In particular, as words were added to the dictionary, it became necessary to save the extended system back to disk as a COM file.

Initially, the required routines were simply hand coded, as in the words PTC, GTC, HOME, and CLS in screen 250 (see the Listing on page 45). These words were enough to install a screen editor written in Forth; but the thought of more hand coding was greeted with something other than joy and enthusiasm. Hand coding is tedious, error-prone, unreadable, and not fun. This was, after all, a personal project.

Arguments are passed to MS-DOS function calls in registers, and any parameters are returned to registers. Forth words which performed the interface to MS-DOS would POP arguments off the stack into appropriate registers, and PUSH registers onto the stack when returning from MS-DOS to Forth. What was needed, therefore, was a convenient syntax and set of words that would compile the required machine code – in effect, a set of macros.

Two special-purpose vocabularies were created: PUSHING and POPPING. In vocabulary PUSHING, the word AX compiles the code for PUSH AX. In vocabulary POPPING, the word AX compiles the code for POP AX. When the DOS function is called, POPPING is invoked. When parameters are returned to Forth, PUSHING is invoked. The following requirements also must be observed:

1. When register SI is used, it must be saved and restored (SI is the instruction pointer in Fig Forth).
2. When AL is returned, AH must be cleared.
3. If any segment registers are used, they must be restored to the CODE SEGMENT.

"When the DOS function is called, the vocabulary POPPING is invoked. When parameters are returned to Forth, PUSHING is invoked."

As implemented, these words assume that one segment is being used. When the MS-DOS function requires a data segment it is simply taken from the DS register. The words can easily be changed to pass data segments from the stack as well. Explanations of the function calls themselves can be found in the IBM Disk Operating System Manual.

The syntax and use of these words are

```
CALL: OPEN_FILE
      PASS( DX )
           0F DOS_FN
      RET( AL ) ;
```

CALL:	A defining word which invokes POPPING, does some initialization of flags, and makes an entry in the dictionary.
OPEN_FILE	The name of the word being defined.
PASS(	Invokes POPPING. This word is not required if no arguments are required by the MS-DOS function. (See GET_DATE, screen 255.)
DX	Compiles code for POP DX.
)	Exits vocabulary POPPING by setting CONTEXT to CURRENT.
0F	The MS-DOS function number.
DOS_FN	Compiles code for MOV AH,0F INT 21H
RET(	Invokes vocabulary PUSHING. This word is required with its terminating '/' even if no registers are returned. (See SET_DTA, screen 257.)
AL	Compiles code for MOV AL,0 PUSH AX
)	Checks flag to see if SI or segments registers need to be restored.
;	Exits vocabulary PUSHING by setting CONTEXT to CURRENT. Compiles code for JMP NEXT.

Figure 1.

David Cornell, 335 Parkside Road, Harrington Park, New Jersey 07640.

best understood by looking at one of them. It is important to note that the word ")" has been redefined in both PUSHING and POPPING, and that the word ";" has been redefined in PUSHING. See Figure 1 (page 44) for an example. In naming Forth words the following conventions were followed:

- {FCB} is a constant holding the address of the active MS-DOS FCB.
- Words with an underscore (_) are primitives; see CREATE_FILE in screen 256.
- Words with a dash (-) use {FCB} and invoke the corresponding primitive; see CREATE-FILE in screen 267.
- Words with a dot (.) are application-level words; see CREATE.FILE in screen 270.

The examples given here do not include all possible combinations. However, once the approach is understood, it should be possible to add whatever is needed. All examples are for DOS function calls. The function calls in the IBM ROM work in a similar manner, using AH to pass the function number. By defining a word that compiles a different interrupt number, the same approach and words can be used for video functions, RS-232 functions, or whatever. See DOS_INT and DOS_FN, in screen 253 for a model.

Two high-level words are included to show how the low-level interface words may be used. ".TODAY" uses GET_DATE and prints the date. SAVE-OBJECT saves the Forth dictionary as a DOS file. It prompts for a filename so that different versions of Forth can be saved on the same disk. The word FREEZE saves some of the USER variables required by COLD. Without this word, the entire dictionary would be saved and loaded; however, COLD would set DP and VOC-LINK to the values they had before new words were added. This definition of FREEZE is for the Fig Forth model, and was taken from the book *All About Forth* by Glen Haydon (available from Mountain View Press, P.O. Box 4656, Mountain View, CA 94040).

Had a Forth assembler been available, the low-level words would certainly have been written as standard CODE definitions. An assembler was not available and, in bringing up new systems, frequently won't be. Under threat of either hand coding or returning to ASM-EXE-CREF-LINK-EXE2BIN, another solution had to be found. The use of the vocabulary mechanism to implement a different syntax is intriguing; and, of course, it further demonstrates the versatility inherent in Forth.

Supplemental Documentation

SAVE-OBJECT
Saves the Forth dictionary as a MS-DOS file. The user is prompted for the filename. The file should be given the extension ".COM" if it is to be loaded and executed by DOS. A warning message is given if the file already exists.

WRITE.MEM
(address #bytes – #bytes.written)
Saves a contiguous block of bytes in the data segment as a DOS file. The user is prompted for the file name. Assumes that the address of the FCB is in constant {FCB}. Sets the record length to 1.

SETUP.WRITE.FILE
Prompts for filename. If the file exists, the user is asked if he wishes to use it. If the answer is "yes," the old file is overwritten. If the answer is "no," exits to QUIT. If the file does not exist, it is created. Assumes that the address of the FCB is in constant {FCB}.

GET.FILE
(– t/f)
Prompts for filename and attempts to open file. "Flag=0" means success (see DOS Manual). Assumes that the address of the FCB is in constant {FCB}.

INPUT.FILENAME
Prompts for filename. Parses filename (no options). Uses PAD and exits to QUIT if error. Assumes that the address of the FCB is in constant {FCB}.

RNDM-BLK-WRITE
(#records – #records.written t/f)
Writes specified number of records. Assumes that the address of the FCB is in constant {FCB} and that the record size has been set in the FCB.

■■J

(Listing begins below)

Forth to PC-DOS Listing (Text begins on page 44)

```
**********************************************************************
***  247  ***
**********************************************************************
00  -->                                                              *
01                                                                   *
02                                                                   *
03              Forth to PC-DOS Interface words                      *
04                                                                   *
05                    David Cornell                                  *
06                    335 Parkside Road                              *
07                    Harrington Park, NJ 07640                      *
08                                                                   *
09                    July 1983                                      *
10                                                                   *
11                                                                   *
12                                                                   *
13                                                                   *
14                                                                   *
15                                                                   *
**********************************************************************
```

```
    ********************************************************************
    ***  248  ***
    ********************************************************************
00  ( misc support: ASCII BKSPACES  Y/N )                               *
01  ( * these words or their equivallents are available in many         *
02      systems )                                                        *
03                                                                       *
04  : ASCII BL WORD HERE 1+ C@ [COMPILE] LITERAL ;  IMMEDIATE           *
05                                                                       *
06  : BKSPACES 0 DO 08 EMIT LOOP ;                                      *
07                                                                       *
08  : Y/N ." (Y/N?)"   ( RETURNS 1=YES  0=NO )                          *
09    BEGIN KEY -33 AND                                                  *
10       ASCII Y  OVER = 2 * SWAP ASCII N = OR -DUP                      *
11    UNTIL  1 - DUP  IF 4 BKSPACES ." ES) "                             *
12                    ELSE 5 BKSPACES ." NO)  "  ENDIF ;                *
13                                                                       *
14                                                                       *
15                                                                 ;S    *
    ********************************************************************

    ********************************************************************
    ***  249  ***
    ********************************************************************
00  -->    ( DOCUMENTATION  JMP.NEXT PTC, GTC, HOME, CLRS, CLS )        *
01                                                                       *
02  ** The address of NEXT must be known to the system.                 *
03     It can be included in the MS-DOS source, or can be               *
04     'found' and included in the dictionary as a constant              *
05                                                                       *
06  JMP.NEXT  compiles a jump to NEXT                                    *
07  PTC  ( row column --- )  PUts Cursor at row, column                  *
08  GTC  ( --- row column )  GeTs Cursor row, column                     *
09                                                                       *
10  HOME   puts cursor at row 0, column 0                                *
11  CLRS   CLeaRS screen, does not affect cursor                         *
12  CLS    CLeas Screen and puts cursor at row 0, column 0               *
13                                                                       *
14                                                                       *
15                                                                       *
    ********************************************************************

    ********************************************************************
    ***  250  ***
    ********************************************************************
00  ( JMP.NEXT  PTC GTC HOME CLRS CLS )                                 *
01  HEX                                                                  *
02  : JMP.NEXT  NEXT HERE 3 + -  0E9 C, , SMUDGE ;                       *
03  : VIDEO.IO  ( VIDEO INTERRUPT ) CD C, 10 C, ;                        *
04  CREATE PTC 58 C, 8A C, D0 C, 58 C, 8A C, F0 C, B4 C, 02 C,           *
05           B7 C, 00 C, CD C, 10 C,  JMP.NEXT                           *
06  : HOME 0 0 PTC ;                                                     *
07  CREATE CLRS B8 C, 0600 , B9 C, 0000 , BA C, 174F , B7 C, 07 C,       *
08     55 C, VIDEO.IO  5D C, JMP.NEXT                                    *
```

(Continued on page 48)

Forth to PC-DOS Listing (Listing continued, text begins on page 44)

```
09  CREATE GTC B4 C, 03 C, B7 C, 00 C,  VIDEO.IO                        *
10      B4 C, 00 C, 8A C, C6 C, 50 C, 8A C, C2 C, 50 C,                 *
11      JMP.NEXT                                                        *
12  : CLS HOME CLRS ;                                                   *
13                                                                      *
14                                                                      *
15                                                          DECIMAL --> *
    ********************************************************************

    ********************************************************************
    ***  251  ***
    ********************************************************************
00  ( POPPING used by PASS( )                                           *
01            HEX                                                       *
02  VOCABULARY POPPING IMMEDIATE     POPPING DEFINITIONS                *
03                                                                      *
04  00 VARIABLE SEG-REG                                                 *
05  00 VARIABLE IPSAV                                                   *
06                                                                      *
07  : ) CURRENT @  CONTEXT ! ;                                          *
08  : IP! IPSAV 3689 , ,  ;                                             *
09                                                                      *
10  : AX 58 C, ;    : BX 5B C, ;   : CX 59 C, ;    : DX 5A C, ;        *
11  : DS 1F C, 1 SEG-REG +! ;      : ES 07 C, 2 SEG-REG +! ;            *
12  : SI IP!  05E C,  4 SEG-REG +! ;  : DI 05F C, ;                     *
13                                                                      *
14                                                                      *
15                       FORTH DEFINITIONS    DECIMAL            --> *
    ********************************************************************

    ********************************************************************
    ***  252  ***
    ********************************************************************
00  (       PUSHING   used by RET(  )                                   *
01  ( closing parentheses and ';' are redifined )                       *
02                                                                      *
03  VOCABULARY PUSHING  IMMEDIATE   PUSHING DEFINITIONS    HEX         *
04                                                                      *
05  : IP@ POPPING  IPSAV 368B , ,  PUSHING ;                            *
06  : AX 50 C, ;    : BX 53 C, ;  : CX 51 C, ;  : DX 52 C, ;            *
07  : DS 1E C, ;    : ES 06 C, ;  : CS 0E C, ;  : AL 00B4 , AX ;        *
08  : FLGS 09C C, ;                                                     *
09  : )  POPPING   SEG-REG @   PUSHING                                  *
10       DUP 1 AND IF CS  POPPING DS   PUSHING   ENDIF                  *
11       DUP 2 AND IF CS  POPPING ES   PUSHING   ENDIF                  *
12           4 AND IF IP@  ENDIF ;                                      *
13                                                                      *
14  : ; CURRENT @ CONTEXT ! JMP.NEXT ;                                  *
15                         FORTH DEFINITIONS   DECIMAL          --> *
    ********************************************************************

    ********************************************************************
    ***  253  ***
    ********************************************************************
00  ( PASS( RET( CALL:  FN# DOS_INT  DOS_FN )                           *
```

```
01                HEX                                                  *
02  : PASS( [COMPILE] POPPING ;                                        *
03  : RET(  [COMPILE] PUSHING  ;                                       *
04  : CALL: CREATE                                                     *
05           POPPING   00 SEG-REG ! ;                                  *
06                                                                     *
07  : FN#    ( c ---     )                                             *
08    B4 C,  ( mov ah i  )  C, ;                                       *
09                                                                     *
10  : DOS_INT  ( int 21 )                                              *
11     CD C,  021 C, ;                                                 *
12                                                                     *
13  : DOS_FN                                                           *
14    FN# DOS_INT ;                                                    *
15                                                          DECIMAL --> *
    ***********************************************************************

    ***********************************************************************
    ***  255  ***
    ***********************************************************************
00  (  GET-DATE   GET-TIME )                                           *
01     HEX                                                             *
02  CALL:  GET_DATE                                                    *
03          02A DOS_FN                                                 *
04         RET( CX DX )  ;                                             *
05                                                                     *
06  CALL:  GET_TIME                                                    *
07           02C  DOS_FN                                               *
08         RET( CX DX ) ;                                              *
09                                                                     *
10                                                                     *
11                                                                     *
12                                                                     *
13                                                                     *
14                                                                     *
15                                                        DECIMAL  --> *
    ***********************************************************************

    ***********************************************************************
    ***  256  ***
    ***********************************************************************
00  (   CREATE_FILE  SEQ_WRITE CLOSE_FILE )                            *
01  ( ***  fcb is assumed to be in the forth segment )                 *
02                              HEX                                    *
03  CALL: CREATE_FILE                                                  *
04        PASS( DX )                                                   *
05           016 DOS_FN                                                *
06        RET( AL ) ;                                                  *
07  CALL: SEQ_WRITE                                                    *
08        PASS( DX )                                                   *
09           015 DOS_FN                                                *
10        RET( AL ) ;                                                  *
11                                                                     *
12  CALL: CLOSE_FILE                                                   *
13        PASS( DX )                                                   *
14          010 DOS FN                                                 *
```

(Continued on next page)

Forth to PC-DOS Listing (Listing continued, text begins on page 44)

```
15        RET( AL ) ;          DECIMAL                              --> *
*****************************************************************

*****************************************************************
***  257  ***
*****************************************************************
00  ( SET_DTA   RNDM_BLK_WRITE )                                     *
01        HEX                                                       *
02  CALL: SET_DTA                                                   *
03     PASS( DX )                                                   *
04       01A DOS_FN                                                 *
05     RET(  ) ;                                                    *
06                                                                  *
07  CALL: RNDM_BLK_WRITE                                            *
08     PASS( CX DX )                                                *
09       028 DOS_FN                                                 *
10     RET( CX AL ) ;                                               *
11                                                                  *
12                                                                  *
13                                                                  *
14                                                                  *
15                                                      DECIMAL  --> *
*****************************************************************
*****************************************************************
***  258  ***
*****************************************************************
00  ( OPEN_FILE   RENAME_FILE   DELETE_FILE )                        *
01          HEX                                                     *
02  CALL: OPEN_FILE                                                 *
03        PASS( DX )                                                *
04            0F DOS_FN                                             *
05        RET( AL ) ;                                               *
06                                                                  *
07  CALL: RENAME_FILE                                               *
08        PASS( DX )                                                *
09            017 DOS_FN                                            *
10        RET( AL ) ;                                               *
11                                                                  *
12  CALL: DELETE_FILE                                               *
13        PASS( DX )                                                *
14            013 DOS_FN                                            *
15        RET( AL ) ;                                    DECIMAL --> *
*****************************************************************

*****************************************************************
***  259  ***
*****************************************************************
00  ( SET_INTERRUPT )                                               *
01                          HEX                                     *
02  CALL: SET_INTERRUPT                                             *
03        PASS(  AX DX )                                            *
04            025 DOS_FN                                            *
05        RET( ) ;                                                  *
06                                                                  *
07                                                                  *
08                                                                  *
```

```
09                                                                      *
10                                                                      *
11                                                                      *
12                                                                      *
13                                                                      *
14                                                                      *
15                                                        DECIMAL   --> *
    ************************************************************************

    ************************************************************************
    ***  260  ***
    ************************************************************************
00  ( DISK_RESET  SELECT_DISK )                                         *
01              HEX                                                     *
02  CALL: DISK_RESET                                                    *
03        0D  DOS_FN                                                    *
04        RET( ) ;                                                      *
05                                                                      *
06  CALL: SELECT_DISK                                                   *
07        PASS( DX )                                                    *
08         0E  DOS_FN                                                   *
09        RET( AL )  ;                                                  *
10                                                                      *
11                                                                      *
12                                                                      *
13                                                                      *
14                                                                      *
15                                                                  --> *
    ************************************************************************

    ************************************************************************
    ***  261  ***
    ************************************************************************
00  ( SEQ_READ                        )                                 *
01                                                                      *
02  ( on return: 1 = no data     3 = partial rec. padded  )             *
03  (            2 = not enough room    00 = ok           )             *
04                                                                      *
05  CALL: SEQ_READ                                                      *
06         PASS( DX )                                                   *
07         014 DOS_FN                                                   *
08         RET( AL )   ;                                                *
09                                                                      *
10                                                                      *
11                                                                      *
12                                                                      *
13                                                                      *
14                                                                      *
15                                                              -->     *
    ************************************************************************

    ************************************************************************
    ***  262  ***
    ************************************************************************
00  ( RNDM_READ   RNDM_WRITE  SET_RNDM_REC )                            *
01                                                                      *
02  CALL: RNDM_READ                                                     *
```

(Continued on next page)

```
03          PASS( DX )                                                    *
04            021 DOS_FN                                                  *
05          RET( AL ) ;                                                   *
06                                                                        *
07  CALL: RNDM_WRITE                                                      *
08        PASS( DX )                                                      *
09           022 DOS_FN                                                   *
10        RET( AL ) ;                                                     *
11                                                                        *
12  CALL: SET_RNDM_REC                                                    *
13        PASS( DX )                                                      *
14           024 DOS_FN                                                   *
15        RET( )     ;                                DECIMAL        --> *
    ************************************************************************

    ************************************************************************
    ***  263  ***
    ************************************************************************
00  ( FILE_SIZE    RNDM_BLK_READ                          )                *
01                        HEX                                             *
02  CALL: FILE_SIZE                                                       *
03        PASS( DX )                                                      *
04         023 DOS_FN                                                     *
05        RET( AL ) ;                                                     *
06                                                                        *
07  CALL: RNDM_BLK_READ                                                   *
08        PASS( CX DX )                                                   *
09         027 DOS_FN                                                     *
10        RET( CX AL  ) ;                                                 *
11                                                                        *
12                                                                        *
13                                                                        *
14                                                                        *
15                                                  DECIMAL         --> *
    ************************************************************************

    ************************************************************************
    ***  264  ***
    ************************************************************************
00  ( PARSE_FILENAME )                                                    *
01  ( bit 0 - scan off leading sep )                                      *
02  ( bit 1 - drive id from parsed line )                                 *
03  ( bit 2 - change file name from parsed line )                         *
04  ( bit 3 - change ext from parsed line )                               *
05              HEX                                                       *
06                                                                        *
07  CALL: PARSE_FILENAME                                                  *
08        PASS( AX DI SI )                                                *
09         029 DOS_FN                                                     *
10        RET( AL ) ;                                                     *
11  ( AL = 01 - ? or * in filename )                                      *
12  ( AL = FF - invalid drive )                                           *
13                                                                        *
14                                                                        *
15                                                DECIMAL           -->   *
    ************************************************************************
```

```
    ********************************************************************
    ***  265  ***
    ********************************************************************
00  ( {FCB} +REC_SIZE +CRRNT_BLCK +CRRNT_REC +REL_REC +FILE_SIZE )  *
01                                                                   *
02    HEX   05C CONSTANT {FCB}  DECIMAL  ( MS-DOS default )          *
03                                                                   *
04                                                                   *
05                                                                   *
06     ( following give offset into ms-dos fcb )                     *
07                                                                   *
08  : +REC_SIZE 14 + ;                                               *
09  : +CRRNT_BLCK 12 + ;  ( block is 128 records      )              *
10  : +FILE_SIZE 16 +  ;  ( **  4 BYTES               )              *
11  : +CRRNT_REC  32 + ;  ( **  1 BYTE  used by seq   )              *
12  : +REL_REC  33 +   ;  ( 2 BYTES used by random    )              *
13                                                                   *
14                                                                   *
15                                                              --> *
    ********************************************************************

    ********************************************************************
    ***  266  ***
    ********************************************************************
00  ( @FILE-SIZE  OPEN-FILE     !REC-SIZE )                          *
01                                                                   *
02  : @FILE-SIZE {FCB} +FILE_SIZE 2@  SWAP ;  (  --- d             ) *
03  : !REC-SIZE  {FCB} +REC_SIZE ! ; (  n ---                      ) *
04  : CLR-FCB {FCB} 36 ERASE ;       ( erase fcb, 0rel/crrnt recs ) *
05  : REL-REC  {FCB} +REL_REC ;      (  --- addr.field            ) *
06  : !REL-REC  SWAP ( lo/hi for 8088 ) REL-REC 2!  ; ( d ---     ) *
07  : CRRNT-REC {FCB} +CRRNT_REC ;    ( --- addr.field            ) *
08  : !CRRNT-REC CRRNT-REC C! ;       ( c ---                     ) *
09  : CRRNT-BLCK {FCB} +CRRNT_BLCK ;  ( --- addr.field            ) *
10  : !CRRNT-BLCK CRRNT-BLCK ! ;      (  n ---                    ) *
11                                                                   *
12  : OPEN-FILE  {FCB} OPEN_FILE ;    (  --- flag                 ) *
13  : CLOSE-FILE {FCB} CLOSE_FILE ;   (  --- flag                 ) *
14  : 0CRRNT-REC 0 !CRRNT-REC  ;      ( set current rec = 0       ) *
15  : 0REL-REC  0 0 !REL-REC  ;       ( set rel. rec = 0  )    -->  *
    ********************************************************************
    ********************************************************************
    ***  267  ***
    ********************************************************************
00  ( CREATE-FILE  RNDM-BLK-WRITE )                                  *
01                                                                   *
02  : CREATE-FILE                                                    *
03     {FCB} CREATE_FILE ;                                           *
04                                                                   *
05  : PARSE-FILENAME ( addr.bytes.to.parse ---          )            *
06                   ( * NO options, see PARSE_FILENAME )            *
07    {FCB} 0 PARSE_FILENAME ;                                       *
08                                                                   *
09  : RNDM-BLK-WRITE  ( #records.to.write --- #records.written t/f )*
10    {FCB} SWAP RNDM_BLK_WRITE ;                                    *
11                                                                   *
12                                                                   *
```

(Continued on next page)

```
13                                                                    *
14                                                                    *
15                                                                 --> *
*********************************************************************
*********************************************************************
***  269  ***
*********************************************************************
00  ( INPUT.FILENAME  GET.FILE )                                      *
01                                                                    *
02                                                                    *
03                                                                    *
04  : INPUT.FILENAME  CLR-FCB   ( uses pad )                          *
05     ." FILENAME: "                                                 *
06          PAD 12 EXPECT PAD                                         *
07          PARSE-FILENAME                                            *
08             IF ." INVALID DRIVE "   QUIT                         *
09             ENDIF ;                                                *
10                                                                    *
11  : GET.FILE   ( --- t/f  )                                         *
12    CR  INPUT.FILENAME  CR                                          *
13      OPEN-FILE  ;                                                  *
14                                                                    *
15                                                                 --> *
*********************************************************************

*********************************************************************
***  270  ***
*********************************************************************
00  ( CREATE.FILE    SETUP.WRITE.FILE )                               *
01                                                                    *
02  : CREATE.FILE                                                     *
03       CREATE-FILE                                                  *
04            IF ." NO ROOM ON DISK  "  QUIT   ENDIF ;               *
05                                                                    *
06  : SETUP.WRITE.FILE                                                *
07     GET.FILE                                                       *
08     IF CREATE.FILE                                                 *
09     ELSE                                                           *
10         ." FILE ALREADY EXISTS - USE IT ? " Y/N CR                 *
11         IF  CREATE.FILE                                            *
12         ELSE    QUIT                                               *
13         ENDIF                                                      *
14     ENDIF ;                                                        *
15                                                                 --> *
*********************************************************************

*********************************************************************
***  271  ***
*********************************************************************
00  ( WRITE.MEM  )                                                    *
01  ( returns #records written )                                      *
02                                                                    *
03  : WRITE.MEM  (  addr  #bytes ---  #bytes.written  )               *
04               (  {FCB} set up   uses GET.FILE      )               *
05    SWAP  SET_DTA  SETUP.WRITE.FILE                                 *
```

```
06     1 !REC-SIZE                                                    *
07     RNDM-BLK-WRITE                                                 *
08       IF ." INSUFFICIENT SPACE ON DISK" ENDIF ( ** DISK ERR )      *
09     CLOSE-FILE                                                     *
10       IF ." DISK CHANGED !!" ENDIF             ( ** DISK ERR )     *
11    ;                                                               *
12                                                                    *
13                                                                    *
14                                                                 --> *
15                                                                    *
    ********************************************************************

    ********************************************************************
    ***  272  ***
    ********************************************************************
00  ( FREEZE )                                                        *
01  ( based on ALL ABOUT FORTH by Glen Haydon )                       *
02  ( The following words valid only for versions of Forth based      *
03    on the FIG Forth listing )                                      *
04                                                                    *
05  : UP                                                              *
06     38 +ORIGIN ;                                                   *
07  : COLD-DATA         ( returns address and count )                 *
08     18 +ORIGIN 16 ;                                                *
09  : INIT-FORTH                                                      *
10     12 +ORIGIN  ;                                                  *
11                                                                    *
12  : FREEZE                                                          *
13     UP  @ 6 +    COLD-DATA    CMOVE                              *
14     ' FORTH 4 + @  INIT-FORTH ! ;                                  *
15  : USER-VERSION  10 +ORIGIN ;                                   --> *
    ********************************************************************

    ********************************************************************
    ***  273  ***
    ********************************************************************
00  ( SAVE-OBJECT )                                                   *
01        HEX                                                         *
02  : SAVE-OBJECT                                                     *
03    FREEZE                                                          *
04    100             ( start address )                               *
05    HERE 100 -      ( #bytes )                                      *
06     WRITE.MEM                                                      *
07     .  ." Bytes written to disk" ;                                 *
08                                                                    *
09                                                                    *
10                                                                    *
11                                                                    *
12                                                                    *
13                                                                    *
14                                                                    *
15                                  DECIMAL                      ;S  *
    ********************************************************************
```

(Continued on next page)

Forth to PC-DOS Listing (Listing continued, text begins on page 44)

```
*********************************************************************
***  275  ***
*********************************************************************
00  ( .TODAY )                                                      *
01        HEX                                                       *
02                                                                  *
03  : SPLT DUP 100 / SWAP FF AND ;  ( split word  n --- c.hi c.lo ) *
04                                                                  *
05                                                                  *
06                                                                  *
07  : .TODAY ( print today's date )                                 *
08     GET_DATE SPLT SWAP                                           *
09       0 <# #S #> TYPE            ( month )                       *
10       0 <# #S 2D HOLD #>  TYPE    ( day )                        *
11       0 <# #S 2D HOLD #>  TYPE    ( yr  )                        *
12     ;                                                            *
13                                                                  *
14                                                                  *
15                                                  DECIMAL  ;S     *
*********************************************************************
```

End Listing

Sorted Diskette Directory Listing for the IBM PC

"This program just might fill a niche left open in the array of PC-DOS utilities. It sorts and formats the directory data and prints the directory to fit inside the diskette envelope."

Just what you always needed! Yet another redundant, duplicative, and space-occupying utility.... This program, however, just might fill a niche left open in the array of PC-DOS utilities. It sorts and formats the directory data and prints the directory to fit inside the diskette envelope. You have to provide your own paper cutter, but the utility does the rest.

The program runs under PC-DOS versions 1.x or 2.0, and will operate on any normal configuration. It does require the IBM or Epson MX80 printer (or equivalent), but it should be easily modified to apply to any printer that offers "condensed" print and eight lines-per-inch line spacing. It is written in Assembler but uses no macros, so either ASM or MASM can be used to assemble the source code. LINK and EXE2BIN are required.

It has two known limitations: It ignores hidden files, and if you're a version 2.0 user and have implemented the "tree" directory paths with multiple directories on a diskette, the subordinate directories will also be ignored. Since the "path" logic seems primarily aimed at fixed disks, and this utility is directed towards diskettes, these limitations should not affect most users.

I wrote the utility after finding myself a frequent player in the following scenario: Phone rings. Voice on the line says, "Say, Dan, you know that program that puts a double whammy on the single-fritz when you've juiced the slobert? My only copy got clobbered, so would you check and see if you've got a backup?" Well, I've got something in excess of 200 diskettes neatly filed in boxes, labeled correctly, etc. The only problem is that a label that says "Miscellaneous Junk Utilities" may have meant something when I wrote it, but it sure doesn't mean much now. My next step is to break into whatever is currently running on the PC and start sticking in diskettes to do a DIR to check.

Actually, incidents like this led to my writing a catalog facility (which does everything but cook) for the PC, but after some months of use, I found that out of a relatively large and complex package I most often used this simple print utility. So I extracted that portion of the big system to pass on to the rest of the world. I'm placing this code in the public domain, so use it as you will.

by Dan Daetwyler

Dan Daetwyler, Route 5, Box 518A, Springdale, Arkansas 72746; (501) 756-0212.

The program is called "COVER" and is composed of seven separately assembled modules: COVER, COTITL, COFREE, COSCAN, COSORT, COPRNT, and COENDP. The listing (page 62) is commented, but I'll include a few words on each of the modules.

COVER is the main control module and contains the major loop. Its first action is to preset the printer to eight lines per inch, condensed print, and to a 44-line page. If you have a different printer, you may need to change the control stream defined at SETPRT. The final zero in the control stream is not sent to the printer but marks the end of string for DOPRT.

Next, COVER checks for the DOS version. The return from the 30H call (which is valid on release 1.x) is the release and version numbers under V2.0 of DOS; on prior releases it returns a zero. This code simply sets a switch that will be tested later. The default drive is determined and saved, and finally the number of drives on the configuration is determined.

COVER then prompts for the drive letter of the drive containing the diskette to be listed. The prompt starts the main loop, which ends after all processing is completed for one diskette. The single character input is first checked for an "escape" (all done, so get off) and next is verified as a legal drive designator. COVER then calls GETTTL (get title), GETFRE (determine free space on diskette), SCAN (load all directory entries into an internal stack), SORT (sequence the stacked entries), and PRINT (format and print the stacked entries). Each of the called routines will be mentioned in the following paragraphs.

When the escape character is sensed, COVER forces a page restore to complete the current listing, resets the default drive to its value on entry, conditionally forces out one more "page" to restore the paper to the true top of form, and finally resets the printer to "power up" status. The control string for the latter is at RESPRT so those using different printers may modify that string if required.

GETTTL is a primitive module that solicits a title from the user. Forty characters are allowed, and the standard DOS function is used to get this string; null titles are permissable. The DOS function is also used to get the system date. DECMAL is a local subroutine that converts the content of the AL to two decimal digits in the AH and AL then stores them in the area defined by the DI register.

GETFRE computes the free space available on the diskette and places the ASCII decimal result in the title line. This module is the reason COVER checked earlier for the DOS version. V2.0 provides a function that gives the free space directly but does not allow you to gain addressability to the File Allocation Table (FAT). V1.x does not have this feature, but it does provide addressability to the FAT. Since this program is to be used in both environments, this code has two paths. The code following the version test is for V2.0. Code starting at the label VERS1 is used to determine free space under the older versions of DOS. VCOM marks the point where the two paths rejoin. If you are only running V2.0, you can decrease the size of the utility by removing the code from the instruction "JMP VCOM" to the instruction before (VCOM). Finally, CONVRT is a public subroutine that converts the AX/DX content to a number with a maximum of six digits, leading zero suppressed, and places the result at the area pointed to by the DI register.

SCAN executes the normal DOS function calls to find an all "wild-card" filename. This process repeats until the entire directory has been scanned. The result is placed in a stack, and a pointer to the start of the entry is placed in a pointer list. The format of the stack entries is the filename.typ, followed by a zero byte and then by the four-byte file

size field from the directory. Since the data transfer area (DTA) has not been reset, this module finds its input in the default DTA at 80H in the program prefix.

SORT is a simple bubble sort of the stacked entries, but it is driven from the pointer list. Yes, I know a shell sort is faster, but for a maximum of 112 entries an old-fashioned bubble sort uses less memory!

PRINT simply uses the sorted pointers to form the final output. The title line has already been formed by the prior modules. The local subroutine PRTENT uses CONVRT to format the file sizes. Four columns will fit within the print size required, so PRINT initially computes column length. An inner loop and an outer loop control the print process. The inner loop prints one entry each execution and is executed four times per line. The outer loop is cycled once per entry in a single column.

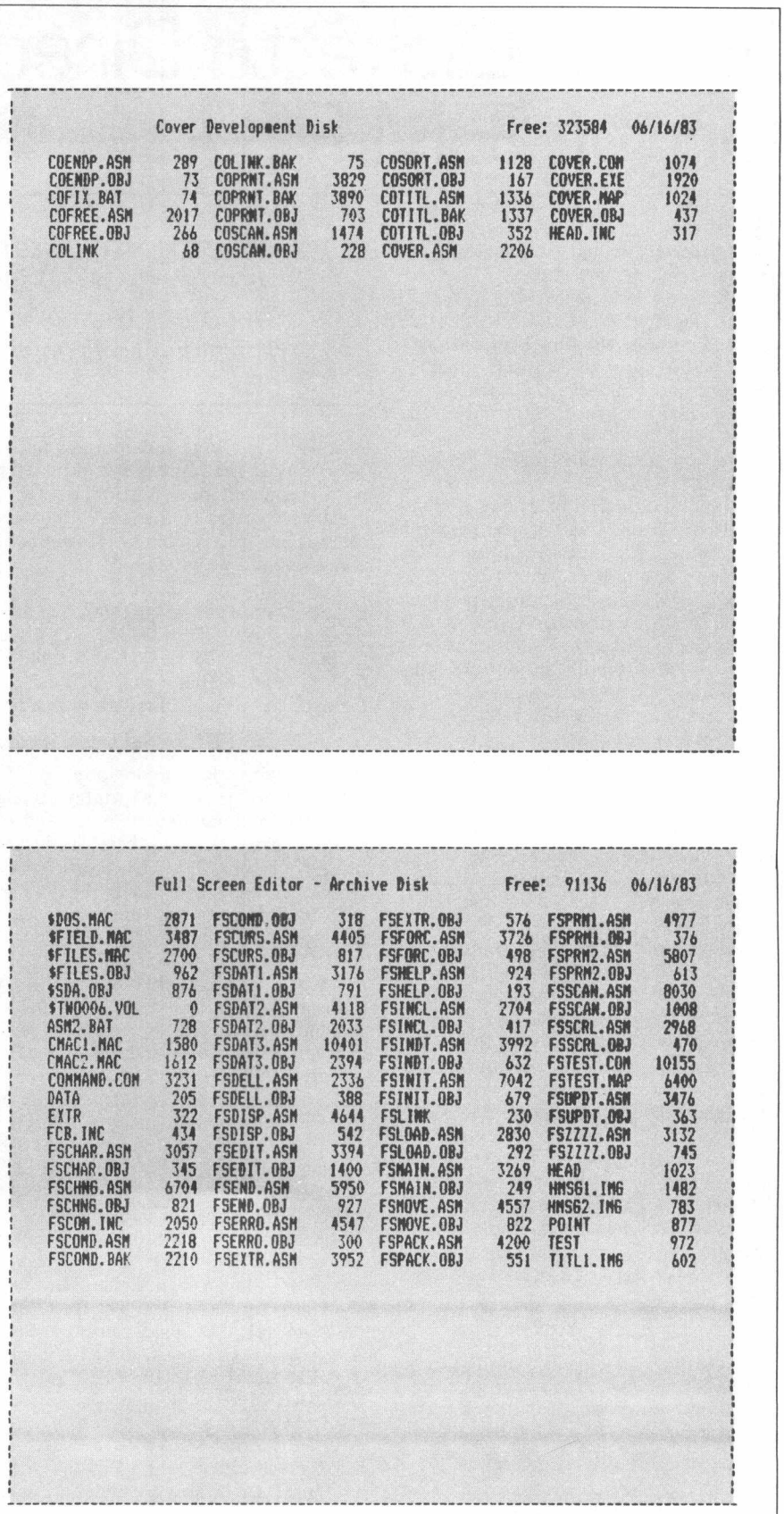
The source module COENDP has no code; it simply provides addressability to the memory above the program code. V2.0 provides this function in a much cleaner manner, but since this utility is to run on either version, this subterfuge is necessary.

That's a very quick walk-through of the code, but with the source comments it should provide sufficient information for any extensions or modifications you may choose to make. Key the source in and assemble the seven modules. The Link control stream is also included in the listing. Be very sure that the module COENDP occurs last in the control stream, as shown in the listing. Otherwise you'll be stacking directory entries over some of the code.

For those of you who are not familiar with COM files, *all* segment registers are preset by the operating system to the same value; although you see no code for setting the DS and ES segment registers, they contain the same value as the CS and SS at program initiation, so change is not required. The stack is automatically allocated at the "top" of the segment (64K) that the program is loaded to, so no stack establishment is required. Finally, the established stack is initialized so that a NEAR return at end of program will return you to DOS correctly, as long as the CS register has not been changed. This program uses the 20H termination call, but could do a simple near RET with the same result.

Link will provide console messages saying that there is no stack segment (that's right) and that there is one error (that's false). This response is normal when you're preparing a COM file. If you get more errors or other indications, you'd better recheck your work.

Now you're ready to run EXE2BIN and then rename the resultant file from COVER.BIN to COVER.COM. That's all



```
                Cover Development Disk                    Free: 323584  06/16/83

COENDP.ASM    289  COLINK.BAK     75  COSORT.ASM   1128  COVER.COM     1074
COENDP.OBJ     73  COPRNT.ASM   3829  COSORT.OBJ    167  COVER.EXE     1920
COFIX.BAT      74  COPRNT.BAK   3890  COTITL.ASM   1336  COVER.MAP     1024
COFREE.ASM   2017  COPRNT.OBJ    703  COTITL.BAK   1337  COVER.OBJ      437
COFREE.OBJ    266  COSCAN.ASM   1474  COTITL.OBJ    352  HEAD.INC       317
COLINK         68  COSCAN.OBJ    228  COVER.ASM    2206
```

```
              Full Screen Editor - Archive Disk          Free:  91136  06/16/83

$DOS.MAC     2871  FSCOMD.OBJ    318  FSEXTR.OBJ    576  FSPRM1.ASM    4977
$FIELD.MAC   3487  FSCURS.ASM   4405  FSFORC.ASM   3726  FSPRM1.OBJ     376
$FILES.MAC   2700  FSCURS.OBJ    817  FSFORC.OBJ    498  FSPRM2.ASM    5807
$FILES.OBJ    962  FSDAT1.ASM   3176  FSHELP.ASM    924  FSPRM2.OBJ     613
$SDA.OBJ      876  FSDAT1.OBJ    791  FSHELP.OBJ    193  FSSCAN.ASM    8030
$TW0006.VOL     0  FSDAT2.ASM   4118  FSINCL.ASM   2704  FSSCAN.OBJ    1008
ASM2.BAT      728  FSDAT2.OBJ   2033  FSINCL.OBJ    417  FSSCRL.ASM    2968
CMAC1.MAC    1580  FSDAT3.ASM  10401  FSINDT.ASM   3992  FSSCRL.OBJ     470
CMAC2.MAC    1612  FSDAT3.OBJ   2394  FSINDT.OBJ    632  FSTEST.COM   10155
COMMAND.COM  3231  FSDELL.ASM   2336  FSINIT.ASM   7042  FSTEST.MAP    6400
DATA          205  FSDELL.OBJ    388  FSINIT.OBJ    679  FSUPDT.ASM    3476
EXTR          322  FSDISP.ASM   4644  FSLINK        230  FSUPDT.OBJ     363
FCB.INC       434  FSDISP.OBJ    542  FSLOAD.ASM   2830  FSZZZZ.ASM    3132
FSCHAR.ASM   3057  FSEDIT.ASM   3394  FSLOAD.OBJ    292  FSZZZZ.OBJ     745
FSCHAR.OBJ    345  FSEDIT.OBJ   1400  FSMAIN.ASM   3269  HEAD          1023
FSCHNG.ASM   6704  FSEND.ASM    5950  FSMAIN.OBJ    249  HMSG1.IMG     1482
FSCHNG.OBJ    821  FSEND.OBJ     927  FSMOVE.ASM   4557  HMSG2.IMG      783
FSCOM.INC    2050  FSERRO.ASM   4547  FSMOVE.OBJ    822  POINT          877
FSCOMD.ASM   2218  FSERRO.OBJ    300  FSPACK.ASM   4200  TEST           972
FSCOMD.BAK   2210  FSEXTR.ASM   3952  FSPACK.OBJ    551  TITL1.IMG      602
```

Figure 1.

it takes. When the utility prompts with a console message asking which driver contains the diskette to be listed, enter the single character drive designator that is appropriate (A-G). No colon or carriage return is required. The program will then prompt for the title you wish to appear on the top line of the list; a forty-character field is provided. If you do not want the title, simply hit the "Enter" key, and the top line of the listing becomes blank except for the date. When the first listing is complete, the utility will loop back and ask for another drive. This iterative process continues until you respond with an escape character to the drive request prompt. Exit is then made to the DOS.

Note that, although the utility prints two envelope covers per page (see Figure 1 on page 61) forms will be advanced to the "top of form" position in effect when the utility was started. The printer will also be restored to "power up" status. If you use Control-Break to exit the utility, the above will not occur, so you'll have to manually realign and reset your printer.

There's not that much code in this utility, but as always, if you don't want to poke this in yourself, send me a diskette with mailer and postage, and I'll copy my source and return it to you. Or if you prefer, send me a check for $7.50, and I'll send you a new diskette (single-surface, eight-sector). You can contact me at the address or phone number that appears with this article.

■■J

(Listing begins below)

Sorted Diskette Directory Listing (Text begins on page 60)

COVER

```
        TITLE   COVER   - Diskette Contents List - Main Module
        SUBTTL  Version 1.0 - June 1983
        PAGE    81,132

;
;                               **********************
;                               *     DD Systems     *
;                               **********************
;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
;
        EXTRN   RESTR:BYTE
        EXTRN   DOPRT:NEAR
;
        ORG     100H
BEGIN:  JMP     START
;
        PUBLIC  VERS
;
VERS    DB      0                       ;DOS Version flag
DDRV    DB      0                       ;Default drive at entry
NDRV    DB      0                       ;Number of drives in system
PRMT    DB      13,10,'Enter drive to list (Esc to quit): $'
ERMSG1  DB      13,10,'Invalid drive$'
SETPRT  DB      27,'O',27,'C',44,15,0   ;Set 1/8" 132 character
RESPRT  DB      27,64,0                 ;Restore to power up status
;
        EXTRN   GETTTL:NEAR,GETFRE:NEAR,SCAN:NEAR
        EXTRN   SORT:NEAR,PRINT:NEAR
        EXTRN   PSX:BYTE
;
START   PROC    NEAR
        MOV     DX,OFFSET SETPRT
        CALL    DOPRT                   ;Set printer
        MOV     AH,30H
        INT     21H                     ;Check DOS Version
        OR      AL,AL
        JZ      NOTTWO                  ;V1.x
        DEC     AL                      ;V2.0
```

(Continued on next page)

Sorted Diskette Directory Listing (Listing continued, text begins on page 60)

COTITL

```
NOTTWO: MOV     VERS,AL                 ;Save DOS flag
        MOV     AH,19H
        INT     21H                     ;Get default drive
        MOV     DDRV,AL                 ;  and save
        MOV     DL,AL
        MOV     AH,0EH
        INT     21H                     ;Get number of drives
        MOV     NDRV,AL                 ;  and save
;
;       Start of main loop
;
MLOOP:  MOV     DX,OFFSET PRMT          ;Prompt for drive
        MOV     AH,9
        INT     21H
        MOV     AH,1
        INT     21H                     ;Get user response
        CMP     AL,1BH                  ;Check for exit
        JE      QUIT                    ;He's done, so get off
        OR      AL,' '                  ;Force lower case
        SUB     AL,'a'-1                ;Compute drive number
        JNC     DRVOK                   ;May be valid drive
ERR1:   MOV     DX,OFFSET ERMSG1        ;Invalid drive message
        MOV     AH,9
        INT     21H                     ;Error message out
        JMP     MLOOP
DRVOK:  CMP     AL,NDRV                 ;Check for installed drive
        JA      ERR1                    ;Drive not installed
        DEC     AL
        MOV     DL,AL
        MOV     AH,0EH
        INT     21H                     ;Make selected drive default
        CALL    GETTTL                  ;Get title
        CALL    GETFRE                  ;Get free space
        CALL    SCAN                    ;Load directory entries
        CALL    SORT                    ;Sequence directory entries
        CALL    PRINT                   ;Produce listing
        JMP     MLOOP
;
QUIT:   MOV     DL,12
        MOV     AH,2
        INT     21H                     ;Force page restore
        MOV     DL,DDRV                 ;Load entry default drive
        MOV     AH,0EH
        INT     21H                     ;Restore default
        TEST    PSX,1
        JZ      NOREST
        MOV     DX,OFFSET RESTR
        CALL    DOPRT                   ;Restore page
NOREST: MOV     DX,OFFSET RESPRT
        CALL    DOPRT                   ;Reset printer
        INT     20H                     ;  and exit
START   ENDP
;
```

```
CODE    ENDS
;
        END     BEGIN
        TITLE   COTITL  - Diskette Contents List - Get Title
        SUBTTL  Version 1.0 - June 1983
        PAGE    81,132
;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
;
        PUBLIC  TITLX,FREE
TPRMT   DB      13,10,'Enter Title: $'
TBUF    DB      40,0                    ;Input buffer for user title
        DB      40 DUP (?)
TITLX   DB      13 DUP (' ')            ;Title line
TIL1    DB      42 DUP (' ')
        DB      'Free: '
FREE    DB      9 DUP (' ')             ;Formatted free space in title line
MONTH   DB      '  /'                   ;Date fields in title line
DAY     DB      '  /'
YEAR    DB      '  ',0                  ;End of title
;
        PUBLIC  GETTTL
GETTTL  PROC    NEAR
        MOV     DI,OFFSET TITLX
        MOV     CX,55
        MOV     AL,' '
        REP     STOSB                   ;Clear title line
        MOV     DX,OFFSET TPRMT
        MOV     AH,9
        INT     21H                     ;Prompt for title
        MOV     DX,OFFSET TBUF
        MOV     AH,0AH
        INT     21H                     ;Get user title
        MOV     CL,TBUF+1               ;Load length
        XOR     CH,CH
        MOV     SI,OFFSET TBUF+2
        MOV     DI,OFFSET TIL1
        REP     MOVSB                   ;Move user title to title line
        MOV     AH,2AH
        INT     21H                     ;Get date
        SUB     CX,1900                 ;Convert to two digit year
        MOV     DI,OFFSET YEAR
        MOV     AL,CL
        CALL    DECMAL                  ;Fill in month/day/year
        MOV     DI,OFFSET MONTH
        MOV     AL,DH                   ;  for title line
        CALL    DECMAL
        MOV     DI,OFFSET DAY
        MOV     AL,DL
        CALL    DECMAL
        RET
GETTTL  ENDP
;
DECMAL  PROC    NEAR                    ;Converts AL to two decimal
        AAM                             ;  digits and stores at SI
        OR      AX,'00'
```

(Continued on next page)

Sorted Diskette Directory Listing (Listing continued, text begins on page 60)

```
        XCHG    AL,AH
        STOSW                           ;Save in image
        RET
DECMAL  ENDP
;
CODE    ENDS
;
        END
```

COFREE

```
        TITLE   COFREE  - Diskette Contents List - Get Free Space
        SUBTTL  Version 1.0 - June 1983
        PAGE    81,132
;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
;
        EXTRN   VERS:BYTE,FREE:BYTE
X10000  DW      10000                   ;Conversion constants
X1000   DW      1000,100,10
;
        PUBLIC  GETFRE
GETFRE  PROC    NEAR
        TEST    VERS,1
        JZ      VERS1
        XOR     DL,DL                   ;Set for default drive
        MOV     AH,36H
        INT     21H                     ;v2.0 - get free space
        MUL     BX
        MUL     CX                      ;AX,DX contains bytes free
        JMP     VCOM                    ;Enter common code
VERS1:  PUSH    DS
        MOV     AH,1BH
        INT     21H                     ;v1.x - get FAT
        XOR     AH,AH
        XCHG    CX,DX                   ;CX has number of units
        MUL     DX                      ;Bytes/allocation unit
        PUSH    AX                      ;Save
        XOR     AX,AX
        MOV     SI,2                    ;First FAT entry
FAT2:   MOV     DI,SI
        SHR     DI,1
        ADD     DI,SI                   ;Compute 1 1/2 bytes
        MOV     DI,WORD PTR [BX+DI]     ;Load FAT entry
        TEST    SI,1                    ;See if odd or even
        JZ      FAT3
        SHR     DI,1
        SHR     DI,1
        SHR     DI,1                    ;Adjust for 12 bits
        SHR     DI,1
FAT3:   AND     DI,0FFFH                ;Three nibbles
        JNZ     FAT4                    ;In use, so don't count
        INC     AX
```

```
FAT4:   INC     SI                      ;Step to next entry
        LOOP    FAT2                    ;Loop through FAT
        POP     CX                      ;Restore bytes/allocation unit
        MUL     CX                      ;Compute total free bytes
        POP     DS                      ;Restore program seg reg
VCOM:   MOV     DI,OFFSET FREE          ;Point to output area
        CALL    CONVRT                  ;Convert size to ASCII
        RET
GETFRE  ENDP
;
        PUBLIC  CONVRT
CONVRT  PROC    NEAR                    ;Converts 6 digits, zero surpressed
        PUSH    DI                      ;Save pointer for later use
        DIV     X10000                  ;Result range 0-99
        AAM
        OR      AX,'00'                 ;Make ASCII
        XCHG    AH,AL
        STOSW                           ;Place in image
        MOV     CX,3                    ;Convert last four digits
        MOV     SI,OFFSET X1000
DIVLP:  MOV     AX,DX                   ;Remainder becomes dividend
        XOR     DX,DX
        DIV     WORD PTR [SI]           ;Power of 10 divide
        OR      AL,'0'                  ;Result range 0-9
        STOSB
        ADD     SI,2
        LOOP    DIVLP
        OR      DL,'0'                  ;Last digit in remainder
        MOV     AL,DL
        STOSB
        MOV     CX,5                    ;Now zero surpress 5 digits
        POP     DI
        MOV     AL,' '
SUPLP:  CMP     BYTE PTR [DI],'0'
        JNZ     DNECVT                  ;Conversion complete
        STOSB                           ;Replace leading zero with blank
        LOOP    SUPLP
DNECVT: RET
CONVRT  ENDP
;
CODE    ENDS
;
        END

        TITLE   COSCAN  - Diskette Contents List - Scan Directory
        SUBTTL  Version 1.0 - June 1983
        PAGE    81,132
;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
;
        EXTRN   PNTR:WORD,SRCE:BYTE
;
NORM    DB      0,'???????????'         ;Dummy FCB for "find/next"
        DB      24 DUP (?)
;
        PUBLIC  SCAN
```

(Continued on next page)

Sorted Diskette Directory Listing (Listing continued, text begins on page 60)

COSCAN

```
SCAN    PROC    NEAR
        MOV     DI,OFFSET PNTR
        XOR     AX,AX
        MOV     CX,120
        REP     STOSW                   ;Clear pointer table
        MOV     BX,OFFSET PNTR          ;BX points to start of pointer list
        MOV     DI,OFFSET SRCE          ;DI points to start of entry stack
        XOR     CX,CX
        MOV     DX,OFFSET NORM
        MOV     AH,11H
        INT     21H                     ;Get first search entry
        JMP     SHORT INNER
LOOP:   MOV     DX,OFFSET NORM
        MOV     AH,12H
        INT     21H                     ;Get next entry
INNER:  OR      AL,AL
        JNZ     DONE
        CALL    SAVE                    ;Stack entry
        INC     CX                      ;Count entry
        JMP     LOOP
DONE:   RET                             ;Returns count in CX
SCAN    ENDP
;
SAVE    PROC    NEAR
        PUSH    CX
        MOV     WORD PTR [BX],DI        ;Save pointer to start of entry
        ADD     BX,2                    ;  and step pointer table reg
        MOV     SI,81H                  ;Point to DTA - file name
        MOV     CX,8
SVLP:   MOV     AL,BYTE PTR [SI]
        CMP     AL,' '
        JZ      NMDNE                   ;End of name
        MOV     BYTE PTR [DI],AL
        INC     SI
        INC     DI
        LOOP    SVLP
NMDNE:  MOV     SI,89H                  ;Point to DTA type field
        CMP     BYTE PTR [SI],' '
        JZ      ALLDNE                  ;No file type
        MOV     BYTE PTR [DI],'.'
        INC     DI
        MOV     CX,3
        REP     MOVSB                   ;Move type field to stack
ALLDNE: MOV     BYTE PTR [DI],0         ;Mark end of string
        INC     DI
        MOV     SI,9DH                  ;Point to size of file
        MOV     CX,4
        REP     MOVSB                   ;  and save in stack
        POP     CX
        RET
SAVE    ENDP
;
CODE    ENDS
```

```
;
        END
```

COSORT

```
        TITLE   COSORT - Diskette Contents List - Sort Entry Stack
        SUBTTL  Version 1.0 - June 1983
        PAGE    81,132
;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
;
        EXTRN   PNTR:WORD,SRCE:BYTE
        PUBLIC  STKCNT
STKCNT  DW      0                       ;Count of entries in stack
;
        PUBLIC  SORT
SORT    PROC    NEAR
        MOV     STKCNT,CX               ;Save entry count
        DEC     CX
        MOV     SI,OFFSET PNTR          ;Point to first stack entry pntr
OUTER:  MOV     DI,SI
        ADD     DI,2                    ;Set to "next" pntr
        MOV     DX,CX
INNER:  CALL    COMPAR                  ;Compare stack entries
        JBE     LEAVE                   ;Ascending sequence, so no change
        MOV     AX,WORD PTR [SI]
        XCHG    AX,WORD PTR [DI]
        MOV     WORD PTR [SI],AX        ;Exchange pointers
LEAVE:  ADD     DI,2
        DEC     DL
        JNZ     INNER                   ;Bubble through inner loop
        ADD     SI,2
        LOOP    OUTER                   ;Bubble through outer loop
        RET
SORT    ENDP
;
COMPAR  PROC    NEAR                    ;This compare always forces short
        PUSH    SI                      ;strings low, since strings are
        PUSH    DI                      ;terminated with nuls.
        PUSH    CX
        MOV     CX,12                   ;Max compare
        MOV     SI,WORD PTR [SI]        ;Point to entry
        MOV     DI,WORD PTR [DI]        ;Point to other entry
        REP     CMPSB                   ;Compare strings
        POP     CX
        POP     DI
        POP     SI
        RET
COMPAR  ENDP
;
CODE    ENDS
;
        END
```

(Continued on next page)

Sorted Diskette Directory Listing (Listing continued, text begins on page 60)

COPRNT

```
        TITLE   COPRNT  - Diskette Contents List - Print Cover Sheet
        SUBTTL  Version 1.0 - June 1983
        PAGE    81,132
;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
;
        EXTRN   PNTR:WORD,STKCNT:WORD,TITLX:BYTE
;
        EXTRN   CONVRT:NEAR
;
        PUBLIC  PSX,RESTR
PSX     DB      0                       ;Pass counter
RESTR   DB      12,0                    ;Printer "restore" forms
DBUF    DB      7 DUP (0)               ;Work buffer for file size
DBLK    DB      '  ',0                  ;Double blank between columns
LFTB    DB      '¦    ',0               ;Left border
RGTB    DB      '    ¦'                 ;Right border (includes CRLF)
CRLF    DB      13,10,0
BCNT    DB      0                       ;Body line counter
;
        PUBLIC  PRINT
PRINT   PROC    NEAR
        INC     PSX                     ;Count numbers of prints
        MOV     BCNT,33                 ;Set body line counter
        MOV     AX,STKCNT               ;Load entry count
        MOV     DH,4                    ;Divide by number of columns
        DIV     DH
        OR      AH,AH
        JZ      SETCNT                  ;Evenly divisable
        INC     AL                      ;Ragged edge

        CBW
SETCNT: PUSH    AX                      ;Entries per column count
        CALL    DOBRDR                  ;Do upper border
        CALL    DOBLNE                  ;  and a blank line
        CALL    DOLFTM                  ;Do left margin
        MOV     DX,OFFSET TITLX
        CALL    DOPRT                   ;Output the title line
        CALL    DORGTM                  ;Do right margin
        CALL    DOBLNE                  ;  and another blank line
        POP     CX
        MOV     BP,CX
        SHL     BP,1                    ;BP contains offset/column in ptr
                                         list
        MOV     SI,OFFSET PNTR          ;Point to start of ptr list
OTLP:   CALL    DOLFTM                  ;Do a left margin
        MOV     DL,4                    ;Set inner loop count to columns
        XOR     BX,BX                   ;Clear column offset reg
INLP:   CALL    PRTENT                  ;Print stack entry
        ADD     BX,BP                   ;Step to next column entry
        DEC     DL
        JNZ     INLP                    ;End of inner loop
```

```
        CALL    DORGTM                  ;Do a right margin
        ADD     SI,2                    ;Step to next ptr
        DEC     BCNT                    ;Decrement body line count
        LOOP    OTLP                    ;End of outer loop
        MOV     CL,BCNT                 ;Load remaining body lines
        XOR     CH,CH
        JCXZ    NOFILL                  ;All used
FILL:   CALL    DOBLNE                  ;Fill out body lines
        LOOP    FILL
NOFILL: CALL    DOBRDR                  ;Do bottom border
        MOV     DX,OFFSET RESTR
        CALL    DOPRT                   ;Restore page
        RET
PRINT   ENDP
;
        PUBLIC  DOPRT
DOPRT   PROC    NEAR                    ;This subroutine simply prints
        PUSH    DX                      ;the string pointed to by the
        PUSH    SI                      ;DX reg on entry.  The string
        MOV     SI,DX                   ;is terminated by a nul byte.
        MOV     AH,5
DPLP:   MOV     DL,BYTE PTR [SI]
        OR      DL,DL
        JZ      PRTEND
        INT     21H
        INC     SI
        JMP     DPLP
PRTEND: POP     SI
        POP     DX
        RET
DOPRT   ENDP
;
PRTENT  PROC    NEAR                    ;Print one stack entry
        PUSH    CX
        PUSH    DX
        MOV     CX,12
        MOV     DI,WORD PTR [SI+BX]     ;DI points to stack entry
        OR      DI,DI                   ;If entry is zero, blank space
        JZ      BLNK1
        MOV     AH,5
PELP:   MOV     DL,BYTE PTR [DI]        ;Print to the end of the
        OR      DL,DL                   ;  name/type entry
        JZ      BLNK2                   ;  blanking remainder of 12
        INT     21H                     ;  character field
        INC     DI
        LOOP    PELP
BACK:   INC     DI
        MOV     AX,WORD PTR [DI]        ;Load file size
        MOV     DX,WORD PTR [DI+2]
        PUSH    SI
        MOV     DI,OFFSET DBUF
        CALL    CONVRT                  ;Convert size to ASCII decimal
        POP     SI
        MOV     DX,OFFSET DBUF
        CALL    DOPRT                   ;Print size
GONE:   POP     DX
```

(Continued on next page)

Sorted Diskette Directory Listing (Listing continued, text begins on page 60)

```
        PUSH    DX                      ;Reload entry value
        DEC     DL
        JZ      PUNT                    ;If last column don't space over
        MOV     DX,OFFSET DBLK
        CALL    DOPRT                   ;Two blanks between columns
PUNT:   POP     DX
        POP     CX
        RET
BLNK1:  MOV     CX,18                   ;No entry, so blank entire column
        CALL    CLER
        JMP     GONE
BLNK2:  CALL    CLER                    ;Blanks remainder of name/type field
        JMP     BACK
PRTENT  ENDP
;
DOBLNE  PROC    NEAR
        PUSH    CX
        CALL    DOLFTM                  ;Output a bordered blank line
        MOV     CX,78
        CALL    CLER
        CALL    DORGTM
        POP     CX
        RET
DOBLNE  ENDP
;
DOBRDR  PROC    NEAR
        MOV     CX,88
        MOV     DL,'-'
        CALL    DLFIL                   ;Output a top or bottom border
        MOV     DX,OFFSET CRLF
        CALL    DOPRT
        RET
DOBRDR  ENDP
;
DOLFTM  PROC    NEAR                    ;Outputs "¦     "
        PUSH    DX
        MOV     DX,OFFSET LFTB
        CALL    DOPRT
        POP     DX
        RET
DOLFTM  ENDP
;
DORGTM  PROC    NEAR                    ;Outputs "     ¦CRLF"
        PUSH    DX
        MOV     DX,OFFSET RGTB
        CALL    DOPRT
        POP     DX
        RET
DORGTM  ENDP
;
CLER    PROC    NEAR                    ;Outputs CX blanks to the printer.
        MOV     DL,' '
DLFIL:  MOV     AH,5                    ;Outputs DL character CX times
CLRLP:  INT     21H
        LOOP    CLRLP
```

```
        RET
CLER    ENDP
;
CODE    ENDS
;
        END
```

COENDP

```
        TITLE   COENDP  - Diskette Contents List - Work Area Definition
        SUBTTL  Version 1.0 - June 1983
        PAGE    81,132
;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
;
        PUBLIC  SRCE,PNTR
;
PNTR    DW      0                       ;Pointer list
SRCE    EQU     PNTR+240                ;Start of entry stack
;
CODE    ENDS
;
        END
```

```
COVER+COTITL+COFREE+COSCAN+COSORT+COPRNT+COENDP
COVER/M
COVER
```

End Listing

Emulate WordStar on TOPS-20

MicroPro's WordStar has long dominated the microcomputer community as a word processing package written to be compatible with Digital Research's CP/M operating system. Christopher Hall's SED is a video editor popular among users of Digital Equipment Corporation's TOPS-20 operating system. This article will describe how to make SED accept certain WordStar commands.

Many microcomputer software firms require large mainframe computers in their development efforts; because of this, software consultants and "job shop" programmers are often required to move from machine to machine like nomads. Regardless of who is doing what, productivity is lost when software development systems have radically different characteristics, but perform the same function. The resulting confusion is seen when, for example, workers enter inappropriate command sequences (sometimes with an aggravating loss of data), or when they must repeatedly stop thinking about what they want to do long enough to remember how to do it.

When analysts began to treat programmed instructions as data, computers became general purpose tools. Changing a machine's function was a simple matter of running a different program. Stored programs made it possible to adapt the tool to the task. Then, when analysts began to treat processing options as data, programs themselves became adaptable. Now, for convenience's sake, we can change a program's characteristics by using different options. With parameter-driven programs, the tool can be adapted to the worker.

SED (Screen EDitor) is a video-oriented character editing program. This versatile utility allows the user to define commands. To configure it for the desired character sequences, the user simply alters its parameter file (SED.INIT).

Some of the differences between SED and WordStar occur when the DEC TOPS-20 operating system traps all

by Steven Fisher

Steven Fisher, Steven Fisher Enterprizes, P. O. Box 457, La Mesa, California 92041.

```
                    Cursor Functions

Redraw Screen               <^B>
Cursor Home                 <PF3>
Cursor Tab                  <^I>, <TAB>
Line Up                     <^E>, <UP-ARROW>
Line Down                   <^X>, <DOWN-ARROW>
Character Left              <^Z>, <LEFT-ARROW>
Character Right             <^D>, <RIGHT-ARROW>
Word Left                   <^A>
Word Right                  <^F>
Screenful Up                <^R>
Screenful Down              <^C>
Start of Line               <ESC ^Z>
End of Line                 <ESC ^D>
Start of File               <ESC ^R>
End of File                 <ESC ^C>
Find String                 <ESC ^F>


                    Editing Functions

Delete Character Left       <^H>, <DEL>, <BS>
Delete Character Right      <^G>
Delete Word Right           <^T>
Delete Line                 <^Y> may use <PF1 nn>, <PF2>, <^K ^B>
Insert Mode On/Off          <^V>
Insert New Line             <^N>
Insert Control Character    <^P>


                    Block Functions

Start of Block              <^K ^B>
End of Block                <^K ^K> for <^K ^C> only
Copy Block                  <^K ^C>
Delete Block                <^Y> after <^K ^B>
Move Block                  <^K ^V> after <^Y>


                    File Functions

Abandon Edit                <^K ESC>
Finish Edit                 <^K ^X>
Save Edit                   <^K ^Z>
Edit Alternate File         <^K ^D>
Read Text Into File         <^K ^R> after <^K ^D>
```

Figure 1.

SED.INIT File Listing

```
/nocase/noitabs/notabs/isave:200/save:1200/reset
/X:TUP,^R:^20(^FRE1($20^RBL)^CU)
/X:TDN,^C:^20(^FRE24($20^RFL)^CD)
/X:CDN,^X:^FRG22($1^RFL)^CD
/X:CUP,^E:^FRL3($1^RBL)^CU
/X:CLF,^Z:^FCE1(^SL)^CL
/X:CRT,^D:^FCE80(^SR)^CR
/X:TAB,^I:^TB
/X:TRT,^F:^IF^E(^LB^CD^XX)^DW^A(^IF^E(^XX)^CR^FCE80(^SR))
          ^DW^X^A(^IF^E(^XX)^CR^FCE80(^SR))
/X:TLF,^A:^FCE1(^SL^FCE1(^CU^FRL3($1^RBL)^LE^XX))^CL^DW^X
          ^A(^FCE1(^SL^FCE1(^XX))^CL)^DW^A(^FCE1(^SL^FCE1
          (^XX))^CL)CR
/X:REWRT,^B:^RW
/X:CASE,^\:^CS
/X:HELP,^J:^HL
/X:FIND,^[^F:^SF
/X:EXIT,^K^X:^XT
/X:READ,^K^R:^WI
/X:DONE,^K^D:^FL
/X:QUIT,^K^[:^AB
/X:SAVE,^K^Z:^SV
/X:CTL,^P:^EC
/X:INS,^V:^IN
/X:IL,^N:^IL
/X:DL,^Y:^DL
/X:DC,^G:^FCE80(^SR)^CR^DC
/X:DW,^T:^IF^E(^LB^CD^XX)^DW^A(^IF^E(^XX)^CR^FCE80(^SR)^DC)
          ^DW^X^A(^IF^E(^XX)^CR^FCE80(^SR)^DC)
/X:YANK,^K^V:$^PT
/X:COPY,^K^C:^PT
/X:BOB,^K^B:^MK
/X:EOB,^K^K:^PK
/X:EOL,^[^D:^IF^E(^LE^XX)^DW^X^E(^LE^IF^X^E(^SR))
/X:BOL,^[^Z:$100^SC^C.(^LB^SL^FCE1(^XX))
/X:BOF,^[^R:$0^GO
/X:EOF,^[^C:$100^GO
/X:100UP,^[A:^FRL3($1^RBL)^CU
/X:100DN,^[B:^FRG22($1^RFL)^CD
/X:100RT,^[C:^FCE80(^SR)^CR
/X:100LF,^[D:^FCE1(^SL)^CL
/X:100F3,^[R:^CH
/X:100F4,^[S:^EX
```

Figure 2.

Redefined SED Functions

X-ON <△Q> and X-OFF <△Q> characters. Wherever WordStar uses <△S>, SED expects <△Z>; also, the <△Q> in WordStar becomes <ESC> in SED. Other differences arise from the dissimilar command structures of the two editors and the limited size of the SED buffer used to define the custom sequences.

The two most frequently used terminals on multi-user DEC computers are the VT-52 and the VT-100. Heath chose VT-52 as their secondary protocol for the H-19 (now Zenith Z-19), while many terminals emulate the VT-100 control sequences – now an ANSI standard.

The SED parameters make the VT-100 directional keys perform as their keytops indicate. Also, the four function keys (<PF1>, <PF2>, <PF3>, and <PF4>) implement the standard SED commands (ENTER, RECALL, HOME, and EXECUTE, respectively). The normal SED control sequences are still available, except where the character has been redefined with a WordStar-compatible function.

Figure 1 (page 76) summarizes the redefined SED editing functions. Figure 2 (at left) lists the initialization file. SED.INIT has no imbedded spaces or control characters (i.e., "△R" is two characters), and space constraints cause long file lines to appear as multiple lines. Every line within SED.INIT begins with a slash (/).

■■J

CP/M Exchange

by Robert Blum

I really enjoy the month of January. It is the only month of the year that I can make as many promises as I want and get away with not keeping them. I think they are called "New Year's resolutions." It's also the time of the year to set new directions. The scope of the CP/M Exchange will be broadened during 1984. There are a large number of CP/M enhancement products that I will be looking at periodically, such as: keyboard code translation modules, spooling programs, and the like. However, this column's main thrust will not change. It will continue as a forum for the exchange of ideas and solutions to problems.

CP/M Plus will also get more attention in the coming months. More manufacturers are beginning to offer it on upgraded systems that have 128K of memory. Televideo's new portable computer, the TP-1, optionally offers 128K and CP/M Plus. It is also starting to show up at the discount software suppliers which will bring it more into the reach of the hacker.

I am now using CP/M Plus on a daily basis with favorable results. It took me a few days to get used to its operational differences; nonetheless, the advanced features make my system much more responsive. So far, I think it is worth the effort and the cost.

A Brief Look Back

1983 has been anything but a composed year in the microcomputer industry. It got underway at a furious pace. Computer stores were opening up on every corner; however, in some areas the demand for small business computers still surpassed the dealers' capacity to supply them. Software-only stores, with walls lined with a never-ending array of games, took their place as an alternate source of advice. Mail order companies continued to slash their profit margins amidst price wars. Software vendors scrambled to convert their wares for the IBM PC while the hardware manufacturers formulated strategies to attract customers away from IBM's unquestioned dominance. This corporate monolith has been trying hard to establish a new standard that all hardware manufacturers must abide by, or run the risk of collapsing under the pressure.

I have always been an S-100 bus user. In my opinion, there is no other architecture that can stave off obsolescence as well, nor match its favorable price-performance factor. It's not uncommon for me to periodically update my system with the newest creations and still be able to get a reasonable price for my old parts.

Aside from all the pragmatic reasons to stay with the S-100 bus, it emits an intoxicating sense of power. To own one means you are a serious computerist. Peeking under the covers at an enormous linear power supply capable of producing several hundred watts of power instills a sense of awe. Completing the effect is the illusion of complexity created by several thousand glinting gold-plated contacts. No faint-of-heart switchers allowed here; this is pure power country! However, I am beginning to feel uneasy because of a definite movement, by some companies that were once dedicated to the S-100 bus, toward today's more traditional single-board architecture.

A New Campus For Godbout U.

I don't think there is any argument about CompuPro being the prominent supplier of S-100 boards. They were the leaders in producing innovative products that kept pace with the chip manufacturers. They also went out on a limb to produce boards that supported some of the newer chips with unknown market potential. They were also the leader with a dual-processor board, attesting to a very bold attitude on their part. All of this took place in an old airplane hanger at the Oakland airport. Unfortunately, the old CompuPro no longer exists. They recently moved into a large office building in Hayward and announced their first single-board computer. As their public relations manager put it, "Bill decided to go legit."

I realize that an image change was necessary for them to compete with the other packaged system manufacturers, and I would be negligent if I did not wish them the best of luck. However, I wonder whether they will remain responsive to the initiated computer users who demand perfection and prefer to integrate their own system. In posing my concerns, I was assured that as long as the board-level portion of their business remains profitable they will continue to actively pursue it; but, to what extent, only time can tell.

The System 10 is a fine example of what CompuPro thinks the future will hold as the state of the art. Gone are the racing stripes across the front of a square 50-pound box. The new package is just high enough to make room for vertical placement of 5-inch disk drives and is painted for the office environment. Inside, it's equally as appealing. An 8088 oversees the operation of four 6MHz Z80s that share common resources and an updated version of MP/M 8-16 that allows transparent execution of both 8- and 16-bit tasks. What does the future hold for them? I overheard mention of some combination of 386s and 286s.

Digital Research To The Rescue

I am admittedly prejudiced against single-board computers, including the IBM PC. Not that there is anything wrong with the PC. It's a fine machine, backed by a company you can depend on; but I generally perceive machines of this type to be limiting.

I was staunch in my beliefs until I recently visited the Dallas branch office of Digital Research. While there, I was fortunate enough to be given a demonstration of Concurrent CP/M with windows. What appeared to be a plain vanilla IBM PC. Not that there is anything wrong with the PC. It's a fine machine, backed microcomputers that I have had the opportunity to use.

We began by bringing up WordStar, my favorite word processing package, in task area one. We then reduced WordStar's screen by two lines, from 24 to 22, to make room for three two-line, 25-character windows to be used by other tasks. Next, dBase II was executed in task area two. As demonstrations go, there were two submit files available to perform various utility tasks such as copying files from one disk to another. These were started in task areas three and four.

After typing in a short letter and saving it, I commanded WordStar to print while I used dBase II. The transition from WordStar to dBase II required only one control sequence. After telling dBase II to find some bogus data, I switched over to task area three to see how things were going there. Again, only one control sequence was necessary. The printer had stopped, so I switched back to WordStar where I was greeted with the "no file" menu. My attention was drawn to window two because dBase II had completed its job and was waiting for another command.

And so it went: readily jumping from one task to another without ever losing track of what was happening in other areas. The only problem I had was keeping the machine busy. This demonstration em-

phasized, however, that the PC's memory-mapped video display is, to a large degree, accountable for this amazing performance. If it had been necessary to wait for an entire screen of data to be written to a terminal at 9600 or 19.2K baud when swapping tasks, much of the system's performance would have been nullified.

Concurrent CP/M is a software package that can transform a machine of modest capabilities into a real workhorse. Given a few months for the current S-100 16-bit offerings to mature and an era of effective computerization will be entered.

PIP's Mystery Device

After all the years that CP/M 2.2 has been poked at, I would have thought that all of its mysteries and secrets had been found. Apparently not. Terje Bolstad of Norway has uncovered a mystery device in PIP. He writes:

"Are you aware that PIP (CP/M 2.2) seems to support an undocumented input device called IRD:?

"Try the command PIP CON:=IRD:.

"The response varies from implementation to implementation, but many systems will hang, continuously writing Ss on the console screen.

"Can any of your readers tell me why PIP supports this device – or is it some old remains unintentionally left in PIP's code?"

Terje's letter was initially routed to Dave Cortesi who verified the presence of the mystery device in his system, and when used, it produced almost identical results as those found by Terje. Dave packaged up his findings and shipped them off to me for further research.

I displayed my PIP.COM file and found that at an offset of approximately 250H, from the beginning of PIP, is a table of valid device names. As reported, the second entry is occupied by a device called IRD:. To find out how it would work on my system I assigned IRD: as an input device directed to a disk file. After a few minutes of disk activity I ended the job and displayed the resulting file only to find it completely filled with DEL (7F) characters.

My interest was piqued by finding a fully functional device that was not described in any of CP/M's documentation. Rather than speculate on its purpose, I wrote a letter to DRI's technical support department in hopes of finding an answer.

Round One

Kathy Welsh, DRI's Technical Communications Representative, promptly responded to my query:

"Upon checking with the analyst for CP/M 2.2, I find that the IRD: device is not standard with each copy of CP/M. Therefore, I suggest you contact your Original Equipment Manufacturer for further information."

I would have accepted her answer except I found it highly unlikely that the same OEM that supplied my system had also produced Dave's, on the West Coast, and another to Terje in Norway.

Round Two

I wrote to Kathy again. This time I explained what we had found and a little background history in hopes that a more complete answer could be found. A few days later I received a phone call from the CP/M 2.2 support analyst – a very nice fellow – who was willing to answer all my questions, except about the mystery device. I was told that IRD:'s presence was proprietary information. Even after I explained that my request was more curiosity than anything else, he remained steadfast in his position.

It seemed that I had come to the end of the road until I remembered that CP/M '83 was coming up soon and DRI would have all their top technicians present. I felt sure that I would be able to bend someone's arm to get my question answered.

Round Three

A few days before the show was to begin, I received a phone call from *DDJ*'s editor to ask if I would have any interest in meeting with representatives from DRI's operating system group. You bet! Not only might I be able to get some insight into DRI's future plans, but I would also have a fresh set of ears to hear my question.

```
00-0F  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
10-1F  00 25 00 E5 0D CC FF FF FF FF FF FF FF FF FF FF
20-2F  00 C5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
30-3F  FF FF 47 FF FF FF FF FF FF FF FF FF FF FF FF FF
40-4F  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
50-5F  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
60-6F  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
70-7F  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Figure 1.

```
A0>
A0>SID
CP/M 3 SID - Version 3.0
#L0
  0000  JMP  EE03
  0003  NOP
  0004  NOP
  0005  JMP  C500
  0008  ??=  20
  0009  LXI  H,F764
  000C  ANA  L
  000D  STAX D
  000E  MOV  D,B
  000F  RNZ
  0010  JP   02E2
#LEE03
  EE03  JMP  EED1
  EE06  JMP  EF61
  EE09  JMP  EF6A
  EE0C  JMP  EF75
  EE0F  JMP  EF55
  EE12  JMP  EF60
  EE15  JMP  EF5E
  EE18  JMP  EF91
  EE1B  JMP  EFA4
  EE1E  JMP  EF9F
  EE21  JMP  F069
#LEF61
  EF61  IN   15
  EF63  ANI  02
  EF65  MVI  A,00
  EF67  RZ
  EF68  CMA
  EF69  RET
  EF6A  IN   15
  EF6C  ANI  02
  EF6E  ??=  28
  EF6F  JM   14DB
  EF72  ANI  7F
#G0

A0>
```

Figure 2.

After explaining my plight to them, I was told that it was doubtful that the answer was proprietary. The question was duly noted and I went away under the assumption that it would be researched and an answer would be forthcoming.

Round Four

Several weeks have slipped by since the show ended, without any word from Pacific Grove, and I am now doubtful that I will hear anything from them. It appears that the only way to solve this mystery is to unravel it ourselves. As an incentive, I will reward the reader who sends in the best explanation and name for the mystery IRD: device with a *DDJ* T-shirt. The award will be made on March 1st and announced in the May issue. The rules are simple: I will judge the entries; in the event of duplicates, the one with the earliest postmark will be selected.

Serial Interfacing

Most of the requests for information that I get are questions on how to interface serial devices not supported by the computer's manufacturer. Most questions are not specifically directed to the serial interface lines themselves but toward the software necessary to drive the computer's ports. And, in many cases, there is even confusion about what port addresses should be used, or what the addresses actually are. To a large degree, this state of confusion is caused by many manufacturers not providing documentation or schematics with their hardware.

The situation is even further compounded when the BIOS source code is also unavailable. When both the hardware and software documentation are missing there is little hope of being able to easily interface a foreign device. Although, if you are willing to invest some time, there are roundabout methods that can be used to find the information you need.

Your first chore will be to create a map of the I/O structure (see Figure 1, page 79). By using the program of Listing One (MAPPER, below), or another of similar design, you will be able to determine what I/O ports exist within your machine. Unfortunately, these results may not be entirely accurate.

Most hardware designs call for inactive signal lines to be held at a high level. In this case, when MAPPER addresses a nonexistent port, the result will be all high levels, or ones. Hence, the resulting display will be FFh. However, this is where the waters really get muddy. Many interface chips contain control registers that cannot be read from. When MAPPER attempts to read from a control register, the result is the same as if it were not present.

Before charging off to map your I/O address space, be aware of one further complication. Some hardware designs specify that when a status port is read from, the status for that port is automatically reset. Since MAPPER indiscriminately reads from all the I/O ports, your machine may quit at any moment – or even worse, start acting strangely. To avoid any disasters I suggest that you remove any vital disks from the system before running MAPPER.

The next step is to take a snapshot dump of the code currently being used to handle an existing device. Either DDT or SID can be used, as both are capable of disassembling instructions directly from memory. I used SID in my example (Figure 2, page 79) but DDT will also produce the desired results.

First, use the L0 command to display the first few instructions, starting at memory location zero. The first instructions displayed will be a jump to the warm start vector in the BIOS jump table. Use only the address portion of this instruction as the operand to the next L command which displays the BIOS vector table, beginning with the warm start vector. Finally, use the address portion of the second instruction as the operand of the next L command. This will disassemble and list the console status routine for you. If it appears that the entire routine is not displayed, another L command without an operand will continue the display from the point where it previously stopped. One further hint: your BIOS may contain Z80 instructions which DDT and SID know nothing about. If you get question marks where the opcode field is normally displayed, use the D command to display the questionable section of memory in hexadecimal form.

You now have almost as much information with the MAPPER listing, which will reveal the I/O port addresses and same steps to disassemble the routines that deal with the other character I/O devices. Then try to correlate this information with the MAPPER listing, which will reveal the I/O port addresses and should show you which bits to use for port control.

Application Note 12

Program: SUBMIT.COM

Products Affected: CP/M V2.2

Copyright © 1982 by Digital Research, Inc. All rights reserved. Used with permission of Digital Research, Inc. Text adapted from original.

Error Description:

SUBMIT does not allow control characters to be a part of the input file. However, any alpha character, either upper or lower case, can be preceded with an up-arrow or caret to designate them as a control character. The following procedure will enable this function.

Procedure:

Make a backup copy of SUBMIT.COM before you use DDT to make the changes shown in Listing Two (page 81).

■■J

(Listings begin below)

CP/M Exchange (Text begins on page 78)

Listing One

```
;
;       MAPPER - a program to display an I/O port map
;

                .z80
                cseg

                .request SYSLIB

                ld      b,16            ;total number of lines to display
                ld      e,0             ;starting port address
```

```
ioloop:
                push    bc              ;save loop count on stack
                call    iomap           ;display one line
                pop     bc              ;restore loop count
                djnz    ioloop          ;loop until done
                jp      Ø               ;warm start
iomap:
                ld      a,e             ;*
                call    pa2hc##         ;call SYSLIB subroutine to output a
                                        ;as two hex characters
                ld      a,'-'           ;display separator character
                call    cout##          ;*
                ld      a,e             ;restore starting port address
                add     a,15            ;last port address for this line
                call    pa2hc##         ;display as two hex characters
                ld      a,' '           ;display blank to separate
                call    cout##          ;*

                ld      b,16            ;ports per line
iomap1:
                push    bc              ;save loop count
                ld      c,e             ;port address to C register
                in      a,(c)           ;input from port
                call    pa2hc##         ;display what it said
                ld      a,' '           ;separator character
                call    cout##          ;display separator
                inc     e               ;adjust for next port to address
                pop     bc              ;restore loop count
                djnz    iomap1          ;loop until complete
                call    crlf##          ;break at lines end
                ret                     ;back until complete

                end
```

End Listing One

Listing Two

```
A>ddt submit.com
DDT V2.2
NEXT PC
0600 0100
-10441
0441  SUI  61
0443  STA  0E7D
0446  MOV  C,A
0447  MVI  A,19
0449  CMP  C
044A  JNC  0456
044D  LXI  B,019D
0450  CALL 02A7
0453  JMP  045E
0456  LDA  037D
0459  INR  A
-s442
0442  61  41
0443  32  .
-g0

A>save 5 submit.com
```

End Listing Two

by Ray Duncan

Floating Point Benchmark

Bill Savage's benchmark testing the speed and accuracy of transcendental functions has drawn more mail than everything else previously published in this column put together. Readers have sent in timings for machines ranging from Apples to the VAX 11-780 (no Cray-I as yet). I will continue to collect the results for publication within the next two or three months; in the meantime, here is a Fortran version of the benchmark contributed by Hugh M. Kawabata of Santa Barbara, California (see Listing One, below). Let's have some translations for less common languages like Ada, APL, and Modula 2!

More on DEC Professionals

Those of you who are interested in purchasing a DEC PC-325 or PC-350 will be happy to hear that my sad report on these computers, published in a recent *DDJ*, is not the end of the story. Since I wrote that column, DEC has sent all PC-350 owners a *free* hardware floating-point processor, upgraded to 512 Kbytes of RAM, and a new version 1.7 of the P/OS hard-disk-based operating system that is considerably faster. A full complement of development tools including an assembler, debugger, and several high-level languages has been released to run "native"on the PC-350. Finally, for those who don't want to deal with the complexity and overhead of P/OS (which is really a very slightly stripped down version of RSX-11M), the RT-11 operating system will be available soon.

All this certainly shows that DEC's heart is in the right place, at least, and the owners of PC-350s with hard disk drives have now got access to all the bits and pieces needed for serious software development. Support for the floppy-disk-only machines is lagging a little behind; however, a product manager for DEC has assured me that they too will soon be totally supported with native tools. I'll report more about this when I've had a chance to use the new software.

Linking Assembler and Microsoft BASIC

Gwen Hummel, of Los Angeles, California, writes: "Enclosed is a short BASIC program that I've found useful. Its essential function is to permit easy one-step linkage for relatively short machine language routines that do not need their own stack segment. The IBM BASIC manual tells the user to run the DOS Debug program to load the routine, then load BASIC and the BASIC application, then execute, alter values, record segment addresses, then BSAVE, and so on.

"This seems so unnecessarily complicated for a short routine, and the enclosed little BASIC program circumvents most of the above. All the user needs to do, after assembling and linking their assembly language portion in the normal manner, is use the EXE2BIN command to convert the .EXE file into a .BIN-type file (this is why no stack segment is permitted, otherwise the EXE2BIN program will not accept the module). The rest is done by the BASIC program." Gwen's program accompanies this column as Listing Two (page 84).

Improved CLINK Utility

Last year we published a small assembly language program called CLINK (*DDJ* No. 79, May 1983) that could be used on the IBM Personal Computer to load custom character set tables, make them permanently resident, and link them to the reserved BIOS table pointer. Patrick Banchy has since sent in a listing of the program converted into a form suitable for the Microsoft assembler and with several significant improvements (see Listing Three, page 86).

He writes: "I have changed the loading of a table into a two-step process. CLINK checks the previous contents of the vector for INT 1FH. If zero, it places CS:0 into the interrupt vector and exits with a DOS interrupt 27H, reserving 1 Kbyte of memory in the process. If the contents of INT 1FH's vector are not zero, it must contain a character table address, which is simply used to set the DTA, then the file is read into memory. The resultant load module is only 231 bytes, and the amount of memory used for the table is precisely 1 Kbyte. No fat on this version!"

■■J

(Listings begin below)

16-Bit Toolbox

Listing One

```
      real a,tan,atan,exp,log,sqrt
      a=1.Ø
      do 1ØØ i=1,2499
            a=tan(atan(exp(log(sqrt(a*a))))) + 1.Ø
1ØØ         continue
      write (1,99Ø)i,a
99Ø   format (' loops=',i3,' Result= ',f8.3)
      stop
      end
```

End Listing One

```
10  'GENERATE.BAS    by G.Hummel
20  '
30  ' This program may be used to organize, generate, and save
31  ' machine code subroutines in forms immediately acceptable to a
32  ' BASIC program.  It is simple and fast to run, pauses to allow
33  ' Shift/Prnt-Scrn for hard copy, and avoids need to use DEBUG.
40  '
50  ' Assemble and link a self-relocatable module without stack segment
51  ' (ignore warning from LINK about stack segment), then use DOS command
52  ' EXE2BIN to convert the .EXE file into a .BIN file.
60  ' Load this BASIC program and change line 190 so that FI$ is assigned
61  ' the name of your input (.BIN) file, then RUN this program.
70  ' Program then reads the bytes from your .BIN file into an integer
71  ' array such that each integer element of the array is unpacked
72  ' (i.e. contains 1 byte from the .BIN file).  Then an assembler
73  ' routine is called to pack the bytes to that each integer element
74  ' holds 2 bytes from the input .BIN file.  On return, there is a
75  ' prompt which offers to write the packed array values to a
76  ' sequential disk file.  If the file is written to disk, it may
77  ' be read into any BASIC program (as an array) and called using
78  ' DEF SEG, VARPTR, and CALL method of accessing the machine code.
100 '
110 ' During execution, program also displays the .BIN file in 3 forms:
120 '           1) original contents of input (.bin) file
130 '           2) input file values as HEX byte values
140 '           3) packed array values generated by this program
150 '
160 ' It is a simple matter to do a Shft-Prtsc to save a hard copy
161 ' of displayed values.  HEX values may be used in READ-DATA type
162 ' of machine code linkage, to "hard-code" the machine routine rather
163 ' than having it reside in a disk file.
170 '
180  CLS:KEY OFF:DEFINT I,X:DIM X(500) 'allows .bin file <=500 bytes
190  FI$="GENERATE.BIN"    'assign to FI$ name of .bin file to be converted
200  GOSUB 460:OPEN FI$ FOR INPUT AS #1  'read bytes into unpacked array X(i)
210  I=0:WHILE NOT EOF(1):A1$=INPUT$(1,1):X(I)=ASC(A1$):A$=A$+A1$:I=I+1:WEND
220  XMAX=I  'xmax=total bytes in .bin file; A$=file as single string of chars
230  CLS:PRINT "Press any key to display contents of your input file:":PRINT
240  B$=INKEY$: IF B$="" THEN 240
250  FOR I=0 TO LOF(1)-1:PRINT X(I);:NEXT:PRINT:PRINT "File has";XMAX;" bytes."
260  CLOSE#1
270 '
280 ' This program accesses its own machine code as a string variable;
281 ' VARPTR is used to get offset of the 3 byte string descriptor;
282 ' bytes 2 and 3 of descriptor are the offset of the string itself;
283 ' CVI function converts offset to a useable integer for CALL stmt.
290 '
300 ZP=0 : DEF SEG : ZP=VARPTR(P$)
305 C$=CHR$(PEEK(ZP+1))+CHR$(PEEK(ZP+2)) : XP=CVI(C$)
310  CALL XP(X(0),XMAX)    'pass array and size of array as parameters
320 PRINT
322 PRINT "Press any key to display input file as "
324 PRINT "hexadecimal byte values (left to right):"
326 PRINT
330 B$=INKEY$:IF B$="" THEN 330
340  FOR I=1 TO XMAX : PRINT HEX$(ASC(MID$(A$,I,1))), : NEXT : PRINT
350 PRINT : PRINT : XMAX=INT(XMAX/2 +.5)    'xmax=new size of array
355 PRINT
360 PRINT "Press any key to display the contents"
```

(Continued on next page)

```
362 PRINT "of the generated array (left to right):"
364 PRINT
370 B$=INKEY$:IF B$="" THEN 370
380 FOR I=0 TO XMAX-1 : PRINT X(I), : NEXT : PRINT
390 PRINT "Press any key to write array values to disk."
400 B$=INKEY$ : IF B$="" THEN 400
410 OPEN "ARRAY.BIN" FOR OUTPUT AS #1
412 FOR I=0 TO XMAX-1
414 PRINT #1,X(I)
416 NEXT : CLOSE
420 PRINT : PRINT "Sequential file ARRAY.BIN has been written to disk."
422 PRINT : PRINT "End of program."
424 END
430 '
440 ' This program reads its machine code routine into a string variable
441 ' rather than using an array (see line 280 for linkage method).  To use
442 ' this method, however, the code must not exceed 255 bytes.  Longer
444 ' routines must use an array.
450 '
460 P$="":FOR I=1 TO 43:READ XB:P$=P$+CHR$(XB):NEXT:RETURN
470 DATA &H55,&H8B,&HEC,&H8B,&H7E,&H08,&H8B,&H4E,&H06,&H50,&HB8
480 DATA &HFF,&HFF,&HF7,&HC1,&H01,&H00,&H74,&H03,&HB8,&H00,&H00
490 DATA &H49,&H8B,&HF7,&H47,&H46,&H46,&HA4,&HE2,&HFC,&HA9,&HFF
500 DATA &HFF,&H75,&H02,&H88,&H05,&H58,&H5D,&HCA,&H04,&H00
```

End Listing Two

Listing Three

```
1                                       name    clink
2                                       page    55,132
3                                       title   'CLINK - Load and Link Graphics Characters'
4                                       assume  cs:cseg
5                               ;
6                               ; CLINK - Load and Link Graphics Character Table
7                               ;
8                               ; Original by Ray Duncan, published in DDJ #74
9                               ; Revised by Patrick Banchy, 1249 Park Ave. #5C, NYC
10                              ;
11                              ; The IBM PC allows the user to define the meanings of the
12                              ; characters in the range 80H-FFH in the graphics modes.
13                              ;
14                              ; This program when first called will allocate the 1 KB of
15                              ; memory needed for the table.  Subsequent calls will load
16                              ; the table specified in the invocation into memory.
17                              ;
18      = 005C                  fcb     equ     05ch    ;default file control block
19                              ;
20      = 0024                  eom     equ     '$'     ;literal ending of string
21      = 000D                  cr      equ     13      ;ASCII carriage return
22      = 000A                  lf      equ     10      ;ASCII line feed
23                              ;
24      0000                    cseg    segment para public 'CODE'
25      0100                            org     100h
26
27      0100                    clink:                  ;entry from PC-DOS
28      0100  33 C0                     xor     ax,ax   ;see if table has been
```

```
29      0102  8E D8                       mov     ds,ax     ;previously allocated
30      0104  BB 007C                     mov     bx,07ch   ;offset of vector
31                                                           ;pick up address of
32                                                           ;table in DS:DX
33      0107  C5 17                       lds     dx,dword ptr [bx]
34      0109  8C D8                       mov     ax,ds
35      010B  0B C2                       or      ax,dx     ;have we been here before?
36      010D  75 18                       jnz     not_1st   ;yes,so read the table
37                                                           ;no,set up table
38      010F  8C C8                       mov     ax,cs     ;address (restore proper
39      0111  8E D8                       mov     ds,ax     ;contents of DS first)
40      0113  33 D2                       xor     dx,dx
41      0115  B4 25                       mov     ah,37     ;using DOS Set Interrupt
42      0117  B0 1F                       mov     al,1fh    ;call
43      0119  CD 21                       int     21h
44                                                           ;tell the operator whats up
45      011B  BA 018F R                   mov     dx,offset nxt_job
46      011E  B4 09                       mov     ah,9
47      0120  CD 21                       int     21h
48                                                           ;save 1 kbytes for the
49      0122  BA 0400                     mov     dx,400h   ;table, terminate but
50      0125  CD 27                       int     27h       ;stay resident.
51
52      0127                        not_1st:                 ;read in graphics table
53      0127  B4 1A                       mov     ah,26     ;first set DTA address
54      0129  CD 21                       int     21h
55      012B  8C C8                       mov     ax,cs     ;restore DS
56      012D  8E D8                       mov     ds,ax
57      012F  BA 005C                     mov     dx,offset fcb
58      0132  B4 0F                       mov     ah,15     ;try and open file
59      0134  CD 21                       int     21h
60      0136  0A C0                       or      al,al     ;does it exist?
61      0138  74 0B                       jz      file_ok   ;yes,proceed
62                                                           ;no,warn operator
63      013A  BA 0163 R                   mov     dx,offset Boo_boo
64      013D  B4 09                       mov     ah,9
65      013F  CD 21                       int     21h
66      0141  B4 00                       mov     ah,0      ;return to PC-DOS
67      0143  CD 21                       int     21h
68
69      0145                        file_ok:                 ;file exists,read table
70      0145  BB 005C                     mov     bx,offset fcb
71                                                           ;set record size = 1024
72      0148  C7 47 0E 0400               mov     word ptr 14 [bx],400h
73                                                           ;set current rec=zero
74      014D  C6 47 20 00                 mov     byte ptr 32 [bx],0
75      0151  BA 005C                     mov     dx,offset fcb
76      0154  B4 14                       mov     ah,20     ;sequential read
77      0156  CD 21                       int     21h
78      0158  BA 0174 R                   mov     dx,offset loaded
79      015B  B4 09                       mov     ah,9      ;tell operator load
80      015D  CD 21                       int     21h       ;was successful
81      015F  B4 00                       mov     ah,0      ;and return to PC-DOS
82      0161  CD 21                       int     21h
```

(Continued on next page)

```
83
84                                      ;
85                                      ; messages for console
86                                      ;
87      0163  0D 0A 4E 6F 20 73         Boo_boo  db       cr,lf,'No such file',cr,lf,eom
88            75 63 68 20 66 69
89            6C 65 0D 0A 24
90      0174  0D 0A 43 68 61 72         Loaded   db       cr,lf,'Character table loaded',cr,lf,eom
91            61 63 74 65 72 20
92            74 61 62 6C 65 20
93            6C 6F 61 64 65 64
94            0D 0A 24
95      018F  0D 0A 4D 65 6D 6F         Nxt_job  db       cr,lf,'Memory and Link for table initialized,'
96            72 79 20 61 6E 64
97            20 4C 69 6E 6B 20
98            66 6F 72 20 74 61
99            62 6C 65 20 69 6E
100           69 74 69 61 6C 69
101           7A 65 64 2C
102     01B7  0D 0A 52 65 72 75                  db       cr,lf,'Rerun to load the table',cr,lf,lf,eom
103           6E 20 74 6F 20 6C
104           6F 61 64 20 74 68
105           65 20 74 61 62 6C
106           65 0D 0A 0A 24
107
108     01D4                            cseg     ends
109
110                                              end      clink
```

End Listing Three

BOOK REVIEWS

PC DOS: Using the IBM PC Operating System

By Ruth Ashley and Judi N. Fernandez
Published by Wiley Press
$14.95, 225 pages
Reviewed by James Moran*

By the time I had finished reading the first chapter of this book, I had decided that it should have been named: "Everything You Ever Wanted to Know about the IBM PC Operating System – and then Some." As soon as I had reached the halfway point in the second chapter, I was getting annoyed at the authors' assumption of the readers' ignorance and the patronizing presentation of information. The book was replete with tidy little packets of information that, once digested, were followed by more, expanded packets. These were, in turn, followed by multiple-choice quiz questions designed to test the readers' knowledge of what he or she had just been spoon-fed. However, by the time I had finished the third chapter, I had decided that this was one of the best books about the IBM PC DOS that I had encountered this year.

The authors don't assume that their readers have minimal knowledge of the disk operating system; they assume *zero* knowledge – and that's exactly what makes this book a good buy for new IBM PC users. Information is presented in digestible quantities and is quickly reinforced with short quizzes, followed by further expansion of the topic. The writing is clear and concise, and the book would make an excellent teaching aid when teamed up with a text on BASIC programming (teachers, take note!).

Those who use the IBM PC as an applications processor and have no interest in programming will find that the book will function quite nicely as an operations manual for the DOS. One nice touch in the book is the scenario method used by the authors to explain error situations: if you receive *this* error message, then *this* is where you went wrong. Another nice touch is the abundance of hands-on examples that readers are encouraged to try on their own PC. Anyone who reads this book and doesn't know how to use the major features of the IBM PC operating system certainly will not be able to lay the blame on the authors.

One note of caution to advanced software types: this book does not delve into the actual construction of the operating system. But then, that was not the authors' purpose. As a manual for newcomers to the IBM PC, or experienced users who never fully understood how to use the features of the operating system, *PC DOS: Using the IBM PC Operating System* succeeds.

Soul of CP/M

By Mitchell Waite and Robert Lafore
Published by Howard W. Sams, Inc.
$18.95 paperback, 391 pages (323 pages of text plus 68 pages of appendixes and indexes)
Reviewed by David W. Carroll

The most popular operating system in the world today is CP/M-80 by Digital Research. It offers flexibility and adaptability and brings large system power to microcomputers. Unfortunately, until recently, programmers have had few learning aids and references for CP/M's internal workings. The technical manuals for CP/M-80 from Digital Research are written at the reference – not tutorial – level and assume a high level of 8080 assembly language programming expertise. To date, most programmers who need to interface with CP/M directly learned from trial and error, from articles in several periodicals (including *Dr. Dobb's*), or from looking at example programs done by others (e.g., the CP/M Users' Group).

In the past year, half a dozen books have been published that offer to take the reader inside CP/M – beyond the applications program and user level – to the machine language function calls and BIOS level. The most recent of these texts is *Soul of CP/M* by Mitchell Waite and Robert Lafore.

The authors are both experienced programmers, and Mitchell Waite, president of the Waite Group, is the originator of the Sams *Primer* computer book series, and co-author of the *CP/M Primer* and *CP/M Bible* and numerous other texts. Robert Lafore is an editor for the Waite Group and president of Interactive Fiction, a game software company.

The book sets out to teach "BASIC or other high-level language programmers" and "assembly language programmers not familiar with the 8080 or the CP/M environment." It tells how to use CP/M's built-in system calls, all about the CP/M disk system, how to "customize" CP/M to work with different basic I/O devices. basic 8080 assembly language, and the use of DDT, LOAD, and ASM. It is important to note that *Soul of CP/M* does not cover the writing, operation, or modification of any aspects of the machine-dependent BIOS interface program beyond simple CRT and printer drivers.

Although the authors sometimes seem simplistic in their approach – for example, using words like "magic" and "secret" to describe some parts of CP/M, and issuing congratulations upon completion of an eleven line program in Chapter 2 – they do present the subject material in a readable, understandable, and usable manner. The reader is led step-by-step through the use of assembly language code to achieve specific results with CP/M. Examples are numerous and, in most cases, actual system output is shown, aiding the first-time user and minimizing publishing errors.

This text is not an assembly language programming manual – the exposure to 8080 code is a by-product of the main goals of the book – but it does provide more than enough support for a high-level programmer to understand what is going on in the examples. The authors refer the reader to other texts for complete assembly language information, although a complete 8080 assembly language reference is included in an appendix.

In order to try out the programs, the reader should be familiar with and have access to an editor or a word processor program. The authors do not attempt to introduce or teach the use of the transient editor (ED) included with CP/M.

The book is organized in a logical manner. It begins by discussing CP/M, its organization and its functions, after which it immediately proceeds into programming illustrations of console system calls using DDT. The authors then introduce the assembler and cover disk operations and the disk directory. Finally, they discuss assembly language module calls for MBASIC and modifications of the BIOS for different peripherals.

While *Soul of CP/M* is, for the most part, a good, accurate reference and training guide, there are several errors which should have been caught before publication of the final text. The book makes several statements which could mislead

*Review copyright © 1983 by Compu-Syn.

an inexperienced reader, and which certainly confuse an experienced one. These inaccuracies may cause the reader to question other facts presented in the text. The introduction states "the Z80 is another chip from Intel." Referring to DDT, *Digital* Research's Dynamic Debugging Tool, Chapter 1 describes "a similar program called MAC" (actually Digital Research's macro assembler) while meaning SID (Symbolic Interpretive Debugger). Other errors included typos in assembly language definitions and "ready to run" code listings. The knowledgeable reader can work through these areas, but those new to assembly code might find some of these mistakes indecipherable.

Another area of weakness in presentation is a lack of well-structured examples (one exception is the BIOS I/O area code in Chapter 9, extracted from a CP/M system from W. W. Components for the IMSAI). Beginning machine code programmers should be exposed to the concepts of macros, libraries, and well-organized code. While understanding that not everyone has M-80 or MAC macro assemblers, the authors could easily have shown an example of a well-designed program or spent a little more time on the examples shown.

The longest program listing is a word-counting routine called WORDS. It was evidently put together out of several smaller modules, each exhibiting a slightly different writing style. The comment at the beginning of the WORDS listing states (to run):

```
; "Type 'scan < filename.typ >',
```

Unfortunately, the program name is "WORDS," not "SCAN." Multiple command lines are used in the source code, such as

```
mvi c,conout! mvi e,cr! call bdos
```

which do not add to the clarity of the example. Also, labels are shown in the listing both with and without a terminating colon, which might further confuse a novice.

All of the major programs listed in *Soul of CP/M* assemble with ASM and run without difficulty. The usable utilities included as examples were a text file display program (similar to the intrinsic TYPE command), a simple text file input program, a text file line counter, a text file word counter, a small space invader type game, a hex-to-decimal converter, and a hex dump program – without ASCII equivalents – (actually a condensed version of the Digital Research DUMP program included with CP/M). I felt that selecting more useful utilities as examples would have better rewarded the effort of keying in all that code. The authors did not include any tips on debugging any errors made in typing in the programs, and did not discuss any of the more common error flags of ASM or MAC.

The authors set out to accomplish a large task and overall have achieved the goals they outlined in the introduction. The book is well designed and attractively laid out with 8080 assembly language tutorial references highlighted in blue. Extensive use is made of assembly code fragments to be run with DDT to teach both CP/M function calls and the 8080 language. It certainly serves as an introduction to using CP/M's built-in assembly language functions and facilities. However, I feel that more care in the final production and editing of this book and a better choice of example programs would have added to its long-term value.

Third Caltech Conference on Very Large Scale Integration

by Randal E. Bryant
Published by Computer Science Press, Inc., 1983, ISBN: 0-914894-84-2
$36.95, hardback, 430 pages
Reviewed by Doug DeGroot, Ph.D.

The field of integrated circuit technology has advanced so rapidly in the last ten years that it is no longer possible to quickly, efficiently, and safely design massively dense integrated circuits without the assistance of advanced, sophisticated software support tools. These tools include programs to assist the chip designer in component placement, performance analysis, timing analysis, and function specification and verification, to name a few.

The Third Caltech Conference on Very Large Scale Integration (VLSI) was devoted to such tools. Held at the California Institute of Technology in March of 1983, the conference was organized into seven sessions. This book contains 22 of the 23 papers presented, organized as follows:

- invited papers (3): these deal with the practical experiences gained in large VLSI projects;
- circuit timing (3): algorithmic approaches to performance analysis and improvements;
- routing and interconnection (3): layout, river routing, and channel routing algorithms;
- formal system models (4): behavioral models of digital systems;
- system building blocks (3): principles of interfacing large collections of VLSI chips to build large systems, including self-timed systems and systolic arrays;
- special-purpose chip architectures (3): includes one on a chess move generator; and finally,
- silicon compilation (3).

These papers are not for the person unacquainted with integrated circuit technology, although they can be read by the non-expert. They represent some of the activities on the leading edge of VLSI technology and research.

■■J

SuperScripsit

Company: Radio Shack
Computer: TRS-80 Models I and III
Price: $199.00

Reviewed by Marlin Ouverson

Scripsit was once a word processing program you didn't hear much about outside the machine-specific publications. Still, with its features, and its ease of learning and use, it ranked among the best and simplest programs for creating, editing, and printing text. Its huge user base also made it one of the most popular.

As a result, thousands were anticipating the long-awaited arrival of *SuperScripsit* last year. So why the delay with this *DDJ* review? The initial release had quite a few bugs in it. True, I used it (carefully) for eight months, working around known defects and calling its programmer when I found new ones. Knowing that these would be corrected and wondering how many more would be discovered, I waited along with other purchasers of the new program until Radio Shack distributed replacement diskettes free of charge in April 1983.

New Features

This review will summarize some of the features beyond those provided by the old Scripsit, touching on the most interesting points. The documentation is generally excellent, failing only in that it follows Radio Shack's policy of not providing enough technical information to allow bit-nibbling hackers to get in and modify the source code.

Menus

Like most sophisticated word processors, this one is menu-driven. Except when customizing the default settings, the menus only present one screenful of options each. And although a good quick-reference card is supplied, there is a Help Menu consisting of seven screens of one-line descriptions of system commands. One nice option allows the user to view disk directory filenames in single-screen format (wake up, TRSDOS designers), although one must still exit to the operating system to use KILL and COPY.

Creating Text

Input of text is straightforward in all regards. Two lines at the bottom of the video screen are reserved for document information. One shows the paper width, margins, tabs, and the actual position each character will have on the printed page. (A character's position on the screen may be offset by displayed control codes and by lack of incremental video spacing when proportional or mono justification is used.) The other status line displays the document name, current page and line numbers, column position, pitch, linespacing, and other values. Thus, while the TRS-80 screen cannot show exactly what the printout will look like, the status lines provide all required data.

SuperScripsit is a large program and does not fit entirely in memory at any one time. But the size of a document, unlike that of the old Scripsit, is limited only by the available space on disk. The switching and overlay techniques employed appear to be clever enough; but to users accustomed to RAM-speed operations, the delays caused by the shuffling process are, subjectively, large. And the bigger the document, the more sluggish all operations in general become. In the later stages of preparing this document, my typing speed surpassed the video update. The "type-ahead" buffer, though, caught the overflow. The occasional missing character was probably due to my own awkwardness at typing without immediate visual feedback.

Editing

Editing is simple with SuperScripsit. The available commands leave only a few keys unused, but are relatively simple to remember. I still miss the old Exchange command of Scripsit, which would quickly switch words or paragraphs. But now one can move or copy blocks to disk for later recall and insertion, move the cursor forward or backward by character (using the arrow keys) or by word, tab, margin, paragraph, screen, page, or to the beginning or end of the file, or to an absolute page or line number, and blocks of text can be "frozen" to prevent accidentally changing their format or contents when executing document-wide commands.

Printing

The user of this package can choose, depending on the capabilities of his or her printer and print wheel, between ragged right margins or justification by proportional or mono spacing. Words can be underscored or double-underscored, in regular or boldface type. Super- and subscripts are available, as are strike through (as in ~~0~~) and top-of-current-form (for printing more than one column per page). These commands are exceptionally simple, and one never needs to leave the current document line to use them.

Programmable User Keys

SuperScripsit allows the user to define up to ten command keys. These can range from the simple (automatically entering often-used character strings) to the sublime (chaining various editing and printing commands). In my first days as an editor of microcomputer publications, before getting my own word processor, I wondered why so many neatly typed manuscripts arrived in hand-scrawled envelopes. The answer was easy to discover: to do this with most programs one must create an entirely separate file consisting of just the name and address formatted for an envelope. For most authors it was easier to just write out the mailing information. With SuperScripsit one can do this from the correspondence document by spending some advance time programming a couple of user keys and then using the software to format the return address flush left, centering the addressee information as desired (and above the date line), and printing the envelope. Then it can move everything back to its original position so that the document can be saved as a boilerplate for future correspondence.

There are simpler uses for the user-programmable keys, such as move cursor to end-of-current-line, delete to end-of-text, continuous scroll until the Break key is pressed, automatic typing of frequently used words or phrases, execute print or other commands, etc. One user key can perform an operation and then call another user key or itself. To a certain extent this helps overcome the fact that each word processor has certain features that make it better for a particular kind of writing. By using these keys, one can customize it to suit one's most common needs.

Proofreading

That SuperScripsit contains a nice proofreading function will be no reassurance whatsoever to those not in possession of the $149.00 Scripsit Dictionary. But people with both have it made. This program handles the task of coordinating its own separate functions – proofing, correcting, and updating a document.

It first checks a document for any words not found in the 73,000 word dic-

tionary (34,000 on Model I); next it presents those words, in context, to the user, who has three options – skip that word, correct it in this instance, or add it to the dictionary's list of "user words." Up to 2,045 words can be added to this list on the Model III – fewer on the Model I, unless you have 48K and enough drives to keep the user list on a disk separate from the dictionary. Then the user is returned to the original document, which has been updated by the program.

Proofread's correction function is delightfully easy to use, so long as one doesn't take the dictionary's documentation too seriously when it tells which commands are at one's disposal (it is, after all, being driven by another program entirely, in this case). Strike-over changes are simple to make, but things can get a little shaky beyond that.

It took 130 seconds for the program to proofread this article (2,166 words) and another 120 for me to do the corrections while "unknown" words were presented one at a time. The program found 59 words it didn't recognize, 11 of which were truly misspelled; the balance (like "SuperScripsit") were either added to the user list, since I employ them often, or were rarely used words I chose to neither change nor add.

The SuperScripsit Proofread function and the Scripsit Dictionary work well. Their limitation is shared by other programs as well: they only check spelling, not usage. A spelling bye and a spelling bee are the same to them; users cannot rely on the software to do the thinking – only a fairly thorough spelling check.

Bugs or Features?

One of the features of this program is the ability to, for example, insert a single-spaced quotation into a double-spaced manuscript. But suppose the user wishes to change the entire document to single-space. By returning to the Open Document Menu he or she can enter the new line space value and return to the document. The status line will increment the line number by one each time the cursor is moved down one line, but the line space indicator will still show that the document is set to double-space and the document will still print that way. In order to get the desired change, the user must define the entire document as a block and then use a block-action command to change from double- to single-space. Only then will the status line and the hard copy match the value requested in the Open Document options.

Block definition can cause some surprises as well. The problem arises when defining the first sentence of a paragraph as a block by using what the manual calls the "quick" technique. On occasion, the defined block will include lines from previous paragraphs. It sometimes treats semicolons as end-of-sentence markers. And if you are marking a one-line paragraph or the last line of a paragraph, the end-of-paragraph marker and the indentation of the subsequent paragraph will be lost. In neither case will any damage be done – if you notice the mistake before, say, deleting the block.

Writers, like runners, sometimes "break through the wall" and type at inspired speeds well above average, if only for short stretches. (Here I am not equating writers with good typists.) Such athletic bursts of manual prowess may, however, confound SuperScripsit or the TRS-80 keyboard monitor. The word "user" may come out "uscr." On experimentation, "uqwe" can appear as "uqwa," "iqwe" as "iqowe," and "oqwe" as "oqiwe." The proofread functions will catch all the preceding errors, but readers should feel free to discover others of their own.

Conclusion

It is well-known that whichever word processor one learns first is one's favorite. Anyone expecting to sit down with SuperScripsit, as with an old friend who has had a facelift, will find a surprise. It is, as stated, a completely new program. Now its complexity is more like that of WordStar, though the mnemonics are far easier to remember. Most of the more advanced features are safely tucked away behind a menu, and it is always possible to extricate oneself from an unwanted situation by pressing the Break key.

SuperScripsit is not bullet-proof. An uneducated user, for example, will sometimes remove a diskette before closing the document he or she is currently working on. In fact, I tried that as a test. It did not entirely destroy the open docudocument, only losing material that had been typed after the last write to disk. What *did* happen, though, was that it inserted portions of the test document near the beginning of the file containing this review, and apparently deleted four pages of work that had followed. Note that this happended to a file whose only crime was that it resided on the same diskette as another file which was treated improperly. If this should happen to you, it may be possible – maybe – to get at the lost text by positioning the cursor on the very first character of the damaged document and issuing a Shift-Down-Arrow command. If you see the missing text, you can edit and print it; whether you can ever join it with the first portion will depend on your patience and ingenuity. It may require less time and effort to just print and delete the last portion, then retype it at its proper location.

But this example falls into the class of operator errors that few programs can entirely work around. Disregarding that type of event, the only real shortcomings of this program arise from limits in state-of-the-art microcomputers – disk drive speed and RAM size. SuperScripsit has earned its place among the top three word processors available for affordable personal computers. The program uses the Radio Shack keyboard and CRT wisely, is friendly without being condescending to experienced users, and at $199.00 is priced just right. Those who think this price is too high probably do not spend a large portion of their time writing and editing textual materials.

COMPAC

Company: Sextant Systems, P.O. Box 251, Holmdel, NJ 07733
Computer: IBM PC with 64K Memory
Price: $49.00

Reviewed by Chuck Crayne

COMPAC is a package which allows the compaction and decompaction of program or data files. There are actually two programs supplied. COMPAC.EXE creates a compacted version of a given file, and DECOMPAC.EXE restores it to its previous form.

Sextant Systems claims a minimum 12.5% reduction in size for an ASCII file with a good chance at 30%. These figures are conservative. In my tests, a 55509 byte documentation file was reduced by 37.2% in about 4 minutes, and a 39936 byte program module (binary file) was reduced by 21.4% in about 3.5 minutes. Decompaction was slightly faster, taking about 3 minutes for the ASCII file and about 2.5 minutes for the binary one.

The documentation is adequate to run the programs, but is a bit pretentious and misleading. For example, it states that the compaction algorithm is based on the repetitive nature of files and that the more repetitive the file, the greater is the expected reduction. This is not true. I constructed a carefully contrived test case consisting of long strings of identical characters. Such a file could have been reduced almost 90% by the use of a fairly common compression technique. Yet COMPAC achieved only a 31.15% reduction – less than that obtained with the standard ASCII file.

Here is how the compaction algorithm actually works. First, it reads the target file and produces a table of all the characters encountered. Then it sorts this table so that the characters appear in order of the number of times each was found in the text. It then assigns bit patterns to each character such that the more common characters can be represented with fewer bits than can the less frequent ones. In this, it is similar to Morse code, in which "E" and "T" are represented with just one dot or dash while some punctuation characters require as many as six.

Because the compaction table could be different for every file, it must be stored with the compacted file. If the size of the compaction table is larger than the space saved by the compaction process – which could happen with a small file – then the compacted file would actually be larger than the original. Should this occur, the program will detect it, and no compacted file will be produced. An appropriate message is issued in this case.

The manual also doesn't bother to point out a couple of drawbacks to compacting your files. First, the compacted file is meaningless to anything except the decompaction program. This means that you are not going to compact any of your active files because you are not going to want to take the time to decompact them every time you want to run a program. Also, you are not going to compact files which you are going to send to someone else unless you know that they also have the decompaction program.

The second problem is that the compacted file is always a binary file even if the original was an ASCII text file. This means that if you compact a file in order to save on telecommunication charges – probably the most likely reason for buying this package – you must use a file transfer protocol which can handle binary files. While such protocols are widely available, there is as yet no universal standard. It is important, therefore, to make sure that you and the intended receiver both have the same transfer technique implemented.

Summary

COMPAC performs as advertised, shrinking large text files by almost 40% and typical binary files by more than 20%. This could provide significant savings in communication costs. Additional savings could be obtained in the costs of media used for archival storage. These potential savings must be balanced against the operational complexities introduced by working with bit-oriented files which must be processed by the decompaction program before they can be used again.

MCDISPLAY

Company: MasterComputing, Inc., P.O. Box 17442, Greenville, SC 29606.
Price $175.00
Computer: CP/M with 8" SSSD drive

Reviewed by Michael Favitta

MasterComputing's (MCI) MCDISPLAY is a software development tool that frees BASIC programmers from the burden of designing hardware-dependent routines to format CRT terminal displays. It performs type checking on input data which simplifies the validity checking done by the application program. A utility program is used to define the displays and save the definitions as CP/M files. Each display consists of a combination of text, messages, prompts, and data entry fields. Each data entry field has one of 19 allowable data types associated with it. The application program calls MCDISPLAY assembly language routines to transmit any of the predefined displays to the terminal, to modify fields on the current display, or to solicit data from the terminal operator. MCDISPLAY programs can be transported to other systems using different terminals without any program changes. All hardware-dependent routines use a terminal definition file that can be customized for any manufacturer's terminal.

An 8080-, 8085-, or Z80-based system running CP/M 1.4 or 2.2 and one 8-inch, single-density disk are required to use MCDISPLAY. A CRT-type terminal with cursor addressability and either Microsoft BASIC-80 or BASCOM are also required. MCDISPLAY uses 6.5K bytes of RAM to store the runtime display management routines, plus some additional memory for a display overlay area.

The MCDISPLAY software package includes a comprehensive user's manual, a terminal definition utility (TERMDEF), a demonstration program that also serves as a tutorial on display definition (DEMO), a display definition utility (DSPDEF), and the display management routines (DSPASM).

Documentation

The MCDISPLAY user's manual comes in a three-ring binder that has labeled dividers separating each section for easy reference. The manual not only looks professional, but is one of the best microcomputer software manuals that I have seen. The manual starts with an overview of the package, followed by a section on each part of MCDISPLAY in the order that you will need them. First, there is a section on TERMDEF, followed by a tutorial on the MCDISPLAY "programming philosophy." Next there is a section on how to use DSPDEF, and finally the DSPASM routines and Microsoft BASIC interface.

The tutorial is the key to understanding MCDISPLAY. MCI not only demonstrates how to use the product, but presents an entire programming methodology. This methodology is based on structured programming and the concept of building program logic around the data entry and display requirements. In the tutorial, you design and code a program using MCDISPLAY. The program you design is actually a modified part of the display definition utility, so you also gain some valuable insight into the internal workings of MCDISPLAY. The source code for this and two other sample programs is included for your experimentation.

The information in the rest of the manual is accurate and complete. Both a table of contents and a usable index are included. Eight appendices that contain useful information, such as file formats and how to resolve some of the common user problems, are also provided. The manual even comes with several types of layout sheets for defining displays, plus a black and white template of each so that Xerox copies can be made.

Terminal Definition Utility – TERMDEF.COM

MCI refers to MCDISPLAY as an "intelligent hardware interface." The intelligence comes from the ability to use MCDISPLAY on any manufacturer's CRT terminal by changing only the terminal definition file. The TERMDF utility is provided to create or modify the terminal definition file. Twenty-two terminals are predefined to TERMDF. If you're lucky enough to have one of them, terminal definition is performed simply by selecting the correct terminal from the menu. If not, you must enter the various terminal control codes needed for the proper operation of MCDISPLAY. The utility can be used to modify existing terminal definitions to change default options.

Terminal definition was the only part of the package that I had any problems with. My terminal was one of the twenty-two, but unfortunately several of the control codes were incorrect. The documentation does make a special point of advising you to review the control codes even if your terminal is predefined. The problem was easily diagnosed, as the demonstration program can be used to verify that your terminal definition is correct.

Demonstration/Tutorial Program – DEMO.COM

Once terminal definition is completed successfully, you can run the demonstration program. The program is a modified version of the display definition utility. Twelve prompts are defined to illustrate the various screen formatting capabilities of MCDISPLAY. The definition of any prompt may be modified and the result viewed.

The demonstration program is set up to do a little advertising for the product. The reason for this is MCI's software return policy. TERMDEF and DEMO are shipped on one disk, while the rest of the package is on another in a sealed plastic envelope. At this point, if you decide that the product isn't for you (and the seal on the envelope is intact), you can return the whole thing and get your money back.

Display Definition Utility — DSPDEF.COM

The Display Definition Utility (DSPDEF) is an interactive menu-driven utility. It is used to create the displays that will be manipulated by calls to DSPASM routines from the application program. The DSPDEF utility is a MCDISPLAY program, using predefined displays as menus and MCDISPLAY data entry routines to input the data.

The first step in the development of a program using MCDISPLAY is to formalize the data entry and display requirements. Up to 99 displays may be defined per application program. All data on a CRT screen is defined as either text, messages, prompts, or entry fields. Text and entry fields are permanent fixed-location attributes of a display, while messages and prompts may appear anywhere on a display and are transitory.

Three types of display layout sheets are provided to aid you in the display definition process. Master layout sheets are representations of the CRT screen. They are used to determine the coordinates of all text and entry fields, as well an any areas reserved for messages or prompts. Message layout sheets are used to define messages, since more than one message may occupy the same location in a display. The third type of layout sheet is for prompts. Prompts are messages with entry fields appended to them. The data entry part of a prompt may be positioned anywhere in the display. This requires another set of coordinates for each prompt in addition to the data type and length for that entry field.

MCDISPLAY routines will not let the terminal operator enter data that does not match the data type specified for that field. This type checking is done as each character is entered. No data is passed tc the application program until one of the six legal input terminators is entered (CR, DEL, Left-, Right-, Up-, or Down-Arrow). There are four numeric and fifteen character data types to choose from.

Once the display layout sheets have been completed, display definition is simply a data entry task using the DSPDEF utility. DSPDEF has the capability to edit and preview any defined display. This allows you to see the displays on the CRT and make layout changes before any code has been written for the application program.

Primitives and Directives — DSPARM.REL

MCDISPLAY uses assembly language routines to transmit predefined displays to a CRT screen. It then manages subsequent data transfers to and from those displays. These routines are divided into two groups, Primitives and Directives.

CATEGORY	NAME	FUNCTION
ERASE	ERSSCN	Erase screen
	ERSEOL	Erase to end of line
	ERSEOS	Erase to end of screen
CURSOR	CURSOR	Position cursor to row/column
	LOCCUR	Return cursor row/column position
WRITE	WRBUF	Position cursor and direct buffer write
	WRSTR	Position cursor and write string
	WRUND	Set field at row/column to underscores
	WRBLN	Set field at row/column to blanks
USER	USER1-USER6	User defined
WARNING	BEEP	Send bell (ASCII 7) to terminal
CONVERSION	FORMI	Integer to output character string
	FORMS	Single precision to output string
	FORMD	Double precision to output string
	DECINT	Input string to integer
	DECSNG	Input string to single precision
	DECDBL	Input string to double precision
ENTRY	ENTRY	Input via field data type and length
	RENTRY	ENTRY with a default value
	RESTORE	Set ENTRY default

Table 1.

CATEGORY	NAME	FUNCTION
SCREEN FORMAT	WRTSCN	Initialize and send display to CRT
ENTRY CONTROL	CLREN	Clear field and set up for entry
	PDFEN	Set entry default and set up for entry
	PRTEN	Position cursor at entry field
	IPTEN	Position cursor and input data
	DEFEN	Print default at field and allow edit
	CHKEN	Return first blank mandatory entry field
PROMPT CONTROL	NEWPR	Write prompt and get response
	CLRPR	Clear prompt from display
	REPPR	Clear response and get new response
MESSAGE CONTROL	WRTMS	Write message to display
	CLRMS	Clear message from display
	PLSMS	Position cursor to message field

Table 2.

Primitives are the less complex functions that relate to the system terminal, data conversion, and data entry. These routines provide a programmer with a great deal of flexibility in controlling the display if desired. The MCDISPLAY Primitives are presented in Table I (page 95).

Directives are the high-level display control routines that use Primitives to manipulate the display. They perform complex display control functions with a minimum amount of programming effort. Most applications use Directives instead of Primitives to reduce design and programming time. The brief descriptions of each Directive (see Table II, page 95) are not intended to be complete. They are provided to give an indication of the types of processing that Directives perform.

The tutorial and sample programs present most of the information needed to use MCDISPLAY Primitives and Directives. Detailed information on the individual routines is presented in the reference sections of the user's manual. These should be studied in some detail before attempting to write programs using MCDISPLAY. Writing MCDISPLAY programs is not as easy as it first might appear. There are two reasons for this.

First, you must interface BASIC-to-assembly-language routines. Simply knowing how the DSPASM routines are called by either BASIC-80 or BASCOM is not sufficient to write MCDISPLAY programs. The programmer must be aware of all the constraints of dealing with assembly language routines from BASIC and be very careful to take them into account when coding. It is easy to make mistakes such as passing a constant instead of a variable or using a variable that is the wrong data type to pass an argument. There is no error checking that will detect these types of errors, and they can cause a variety of symptoms, the worst of which is a system crash. The user's manual and tutorial do contain a comprehensive discussion of the pitfalls that are inherent in a BASIC-to-assembly-language interface. If you read the manual and follow the prudent coding rules that are suggested, you shouldn't have any problems.

Second, writing "user friendly" programs that provide full CRT screen capability is not trivial, even with the help of a powerful tool such as MCDISPLAY. The tutorial presents a program design philosophy for use with MCDISPLAY. The philosophy is to create structured programs which have the main control logic built around the data entry and display requirements. A program that implements this philosophy is designed and coded in the tutorial. This program is intended to be used as a framework for your own application programs in order to speed development. At the same time it guides your program development towards the proposed design philosophy. Included in the sample program is a methodology for controlling cursor movement between the various entry fields on a display. Using this methodology, the user may enter the data in any order by moving from field to field via the six legal field terminators. A cursor movement control array is used to define what field will receive the cursor on each of the possible entry terminators. This style of data entry adds to the user friendliness of a program. Since multiple displays, prompts, and messages don't take up any more memory, there is no reason why extensive prompting and help displays can't be used to increase program usability.

MCI doesn't just leave you hanging with a theoretical presentation on the use of structured programming. Methods for building blocked IF-THEN-ELSE and CASE structures using Microsoft BASIC are detailed in the tutorial. MCI has also released a companion product called TRANSFORM for $39.95 (reviewed in the December 1983 issue of *DDJ*), which was developed to implement the structured approach to program development that is presented in the MCDISPLAY tutorial. It turns Pascal-like code into code that can be run by Microsoft or Microsoft-compatible BASIC's. A review of TRANSFORM will be forthcoming.

Interface for MCBASIC and BASCOM

MCDISPLAY is designed so that both MBASIC and BASCOM may be used with minimal program-changing. All DSPASM routines are accessed through a vector table that is present at the beginning of the module. A relocating loader (DSPINT) is provided for use with MBASIC to move DSPASM into memory directly under CP/M. The loader also loads and executes MBASIC. In order to use the CALL statement from MBASIC, the entry points for the DSPASM routines must be defined. A BASIC routine is provided to calculate the base of the vector table as loaded by DSPINT along with the entry point of each DSPASM routine. The only change required to compile the MBASIC version of a program with BASCOM is to remove or comment out the DSPASM entry point table. The DSPASM module is then linked into the BASIC program along with a module to reserve display overlay space using Microsoft's Link-80.

Reviewer's Comments

I was impressed by how complete this software package is, especially the documentation. The documentation provides reference material and detailed information on how to use the package most effectively from the programmer's perspective. The information is presented in steps, so even novice programmers can use this sophisticated software development tool.

MCDISPLAY delivers what it advertises. It reduces the time required to develop application programs that are user-friendly and self-instructing. Both these objectives are achieved through the use of CRT-formatted data entry and display. These displays are not restricted by the available memory in the host computer. The one negative aspect of the product is that the programmer must deal with the intricacies of a BASIC-to-assembly-language interface. MCI has minimized the impact of this interface through the documentation and the relocating loader. I think the usefulness of this software tool outweighs this disadvantage.

Like any software package, MCDISPLAY is not for everybody. Simple programs or those with very little data entry requirements can certainly be developed just as effectively (but perhaps not as eloquently) without MCDISPLAY. Large application programs with multiple data entry requirements, however, will quickly return the $175 investment in increased productivity of programmers and end users. If you have any doubts about the product, you can review the user's manual and try out the demonstration program. With MCI's software return policy, the most you can lose is the cost of shipping the product back.

The only hard error I found in the package was with the terminal definition utility. I called MasterComputing about the incorrect control codes for my terminal and found them very receptive. They alerted me to several problems that had been reported, but all were hardware related and did not apply to my system. I used the opportunity to inquire about the future of the product and found out that they are currently working on version 2. It will contain a number of enhancements, the most notable of which is the ability to define user field types via a legal character mask.

Although display formatting utilities have been available for years on larger computer systems, to my knowledge this is the first product of its kind for CP/M-compatible systems. I congratulate MCI for a job well done on their first entry into the software market.

SMAL/80

Company: Westico, 25 Van Zant Street, Norwalk, CT 06855
Computer: CP/M 2.x or greater
Price: $153.00 ($178.00 for the Z80 version)
Reviewed by Dr. Greg Louis

It is a truth generally acknowledged

Intel	Zilog	SMAL
MOV B,A	LD B,A	B = A
MOV A,M	LD A,(HL)	A = M(HL)
LDAX D	LD A,(DE)	A = M(DE)
MVI D,DATUM	DL D,DATUM	D = DATUM
LXI D,ADDR	LD DE,ADDR	DE = ADDR
XCHG	EX DE,HL	DE <=> HL or HL <=> DE
INR H	INC H	++H
INX H	INC HL	++HL
CMA	CPL	A = NOT A
CMC	CCF	CY = NOT CY
CMP M	CP (HL)	F = A : M(HL)
XRI 80H	XOR 80H	AF = A XOR X'80'
RLC	RLCA	ACY = A ROTATE LEFT n
RAR	RRA	ACY = ACY ROTATE RIGHT n
	EXX	BC DE HL <=> (BC DE HL)
	LDIR	LOOP
		M(DE)=M(HL),++DE,++HL,--BC
		REPEAT UNTIL BC ZERO

Figure 1.
Examples of Intel, Zilog, and SMAL instructions.

"Budget cuts!"

Danby

that the benefits of assembly language in speed and compactness are often outweighed by two primary difficulties: debugging and maintenance. The usual prediction is that it takes six times as long to debug a substantial assembly language program as it did to write the code in the first place. Making significant changes to a running program is also far more difficult than if the program were written in high-level code. The question is: need this be so?

This problem has three main causes. One, the program size that results from 1:1 correspondence between source and machine instructions is unavoidable (modular construction, however, can do much to limit trouble from this source). The second is the relatively incomprehensible nature of the mnemonics in which most assembly languages are written; and the third is the impossiblity of expressing structure explicitly in the source code (except by liberal use of comments). SMAL/80 represents an attempt at an assembler that does not suffer from these last two defaults.

Currently advertised under the rubric, "Help Stamp Out Mnemonic Plague," the package consists of two programs: a general macroprocessor and an 8080 (or Z80) assembler with structured flow-control pseudo-ops. The latter, which includes IF..THEN..ELSE and LOOP..REPEAT (the REPEAT can take a WHILE clause), compiles to appropriate jump instructions. Special handling for IF..GOTO, IF..CALL, and IF..RETURN ensures that these pseudo-ops do not generate overhead. IF ZERO RETURN; compiles like the Intel instruction RZ, while IF ZERO THEN RETURN; would compile like JNZ $+4 followed by RET. SMAL/80 was written as a serious attempt to make assembly code more legible, but the claim that it has eliminated mnemonics is exaggerated. The examples of Intel, Zilog, and SMAL instructions in Figure 1 (page 98) will make this clear. (Note that n in the rotate examples may range from 0 to 7, and that the appropriate number of rotate opcodes will be compiled.)

Enormous power is provided by the macroprocessor. For example, I find the FOR construct presented in Kernighan and Plauger's book *Software Tools* to be a flexible, congenial way of defining a loop. The construct is

```
FOR([initialization] <> [condition]
    <> [step]) [statement]
```

which translates as follows:

```
initialization;
WHILE condition DO {statement; step};
```

Here is the macro definition for SMAL/80. Note that the first character after the word .DEFINE serves as delimiter for the template and the expansion:

```
.DEFINE:FOR ([initialization]
    <> [condition] <> [step])
```

SMAL/80 Review (Text begins on page 98)

Listing One

```
/*   FILE:   VERSION.SRC
             DATE:   820522
             FOR:    Typing first line of .PRN file(s)
             (pseudocode version)
*/
             TITLE = 'Version - check latest compilation'

/*   routine and constant equates          */
             defdma is at X'80'   /* CP/M default i/o */;
             deffcb is at X'5C'   /* CP/M default fcb */;
             warmboot is at 0     /* CP/M restart addr */;
             bdos is at 5         /* CP/M utility call */;
             formfeed is X'0C';
             newline is X'0A';
             outchr is 2          /* CP/M out char */;
             openfl is 15         /* CP/M open file */;
             search1 is 17        /* search for first */;
             searchn is 18        /* search for next */;
             readseq is 20        /* read sequential */;

             ORIGIN X'100';
             start:
             Copy deffcb to fcb1;
             set ext=prn;
             FOR (first match<>match found<>next match)
                find filename;
                copy filename to fcb2;
                open from fcb2;
                read one sector from fcb2;
                skip past formfeed;
                write till newline;
                <>
             GOTO warmboot;
             fcb1: RESERVE 33;
             fcb2: RESERVE 33;
             version MODULE NAME start;
             END PROGRAM start;
```

End Listing One

Listing Two

```
(This first macro redefines the macro definition format!)
                .DEFINE%
                DEFINE
                [template]
                AS
                [expansion]
                ENDM;
                %.DEFINE|[template]|[expansion]|%
```

(Continued on next page)

[statement] <> : [initialization] ;
LOOP
IF NOT [condition] BREAK;
[statement] [step] ;
REPEAT:

These two additional definitions handle special cases that may be produced in the macro expansion:

.DEFINE:*/;:*/:
.DEFINE;NOT NOT : :

The power of the microprocessor is not limited to merely adding new constructs to the assembler. Actually, the macroprocessor can be useful alone (e.g., for initial typing of a long text where a given string appears many times over, or as a preprocessor to another high-level language). The example in Listing One (page 99) shows how it is possible to formalize the top-down design of a program. The application is to print out the first line of any .PRN files specified by the user in the command line. This first line, with Digital Research's MAC, Microsoft's MACRO-80, and SMAL/80, can easily be made to contain timestamp information.

Listing Two (page 99) shows the macros that turn the pseudocode above into legal SMAL/80 source. As a general comment, I apologize for the over-utilization of FOR loops in these definitions, many of which are unnecessarily complicated by the need to clear the Z flag as part of the initialization. These could have been written more simply as

LOOP. . REPEAT WHILE NOT ZERO;

but I was anxious to give the FOR macro a workout.

Finally, Listing Three (page 102) gives the resulting code in "legal" SMAL/80. The code is exactly as translated by the macroprocessor. Its legibility could be improved by pretty-printing; however, the documentation provided by the pseudocode source and macro definitions makes this step unnecessary for most purposes.

The SMAL/80 assembler can produce either an Intel hex file or a relocatable object code as input for linkage. The SMAL users' group was reported last year to be nearly ready to publish a Microsoft .REL to SMAL format translator, but I have heard nothing since despite my year's free membership that was said to be included in the SMAL/80 purchase price.

The speed of compilation is satisfactory, even making allowance for the necessity of running the macro preprocessor. The command format for input to both programs is clumsy – all command information for a given run must appear on the CP/M command line! Users of SMAL/80 would be well advised to obtain Super-Sub, or a similar fancy SUBMIT replacement such as Richard Conn's SUB or ZEX.

SMAL/80 provides a neat combination of the compactness, power, and speed of assembly language and the legibility and modular structure of high-level languages. Nice! It's available with what the authors call the "ultimate demo": 45-day, no-questions, money-back guarantee. (The macroprocessor is available separately for $78 and *everyone* except diehard BASIC programmers should have one – it's a powerful tool.) The two main reasons for *not* using SMAL/80 as an all-purpose assembler are: (1) the output is not Microsoft .REL-compatible; and (2) SMAL/80 achieves legibility at the cost of more typing, which will not be appreciated by the two-fingered. ■■J

(Listings begin on page 99)

SMAL/80 Review (Listing cont., text begins on page 98)

Listing Two

```
(define equates)
DEFINE
is
AS
EQU
ENDM;

DEFINE
EQU at
AS
EQU
ENDM;

(next convert label)
DEFINE
pseudocode
AS
SMAL/80
ENDM;

(next convert pseudocode to SMAL)

DEFINE
Copy deffcb to fcb1
AS
FOR (HL = deffcb; DE = fcb1; A = 9; AF = A OR A; C = A
   <>NOT ZERO<>++HL; ++DE; --C)
   A = M(HL); M(DE) = A;
   <>
ENDM;
(the A=9...C=A above is just to clear the Z flag)

DEFINE
set ext=prn;
AS
HL <=> DE; M(HL) = 'P'; ++HL; M(HL) = 'R'; ++HL; M(HL) = 'N';
ENDM;

DEFINE
first match
AS
DE = fcb1; C = search1; CALL bdos; AF = A OR A
ENDM;
DEFINE
match found
AS
POS
ENDM;



(look for another file that matches the spec)
DEFINE
next match
```

```
AS
getnext:
/* first blank out fcb2 from 12 to 33 */
FOR (DE = fcb2+12; A = 21; AF = A OR A; C = A; A = 0
   <>NOT ZERO<>++DE; --C)
   M(DE) = A;
   <>
/* next reset the drive byte */
A = M(fcb1); M(fcb2) = A;
/* find the file we just printed */
DE = fcb2; C = search1; CALL bdos;
/* copy the (possibly ambiguous) filename from fcb1 */
FOR (HL = fcb1; DE = fcb2; A = 9; AF = A OR A; C = A
   <>NOT ZERO<>++HL; ++DE; --C)
   A = M(HL); M(DE) = A;
   <>
/* search for another match and set M if none */
DE = fcb2; C = searchn; CALL bdos; AF = A OR A
ENDM;

(use the directory code to get the entry in the default buf)
DEFINE
find filename;
AS
ROTATE A LEFT 5; A = A + X'80'; L = A; H = 0; ++HL;
ENDM;

DEFINE
copy filename to fcb2;
AS
A = M(fcb1); M(fcb2) = A;
FOR (DE = fcb2+1; C = 11<>NOT ZERO<>++HL; ++DE; --C)
   A = M(HL); M(DE) = A;
   <>
/* Zero rest of fcb */
FOR (A = C; ++C; C=21<>NOT ZERO<>++DE; --C)
   M(DE) = A;
   <>
ENDM;

DEFINE
open from fcb2;
AS
DE = fcb2; C = openf1; CALL bdos;
IF A = A OR A NEG GOTO getnext;
ENDM;

DEFINE
read one sector from fcb2;
AS
DE = fcb2; C = readseq; CALL bdos;
IF A = A OR A NOT ZERO GOTO getnext;
HL = defdma;
ENDM;

DEFINE
skip past formfeed;
AS
```

(Continued on next page)

Listing Two

```
LOOP
   A = M(HL); ++HL;
REPEAT WHILE A:formfeed NOT EQUAL;
ENDM;

DEFINE
write till newline;
AS
LOOP
   E = M(HL); C = outchr;
   PUSH HL; CALL bdos; HL = POP;
   A = M(HL); ++HL;
REPEAT WHILE A:newline NOT EQUAL;
ENDM;
```

End Listing Two

Listing Three

```
/*   FILE:   VERSION.SML
     DATE:   820522
     FOR:    Typing first line of .PRN file(s)
             (SMAL/80 version)
*/
     TITLE = 'Version - check latest compilation'

/*   routine and constant equates                */
     defdma EQU X'80'          /* CP/M default i/o */
     deffcb EQU X'5C'          /* CP/M default fcb */
     warmboot EQU 0  /* CP/M restart addr */
     bdos EQU 5                /* CP/M utility call */
     formfeed EQU X'0C';
     newline EQU X'0A';
     outchr EQU 2              /* CP/M out char */
     openfl EQU 15             /* CP/M open file */
     search1 EQU 17            /* search for first */
     searchn EQU 18            /* search for next */
     readseq EQU 20            /* read sequential */

     ORIGIN X'100';
     start:
     HL = deffcb; DE = fcb1; A = 9; AF = A OR A; C = A
   ;
     LOOP
     IF ZERO BREAK;
   A = M(HL); M(DE) = A;
   ++HL; ++DE; --C;
     REPEAT;
     HL <=> DE; M(HL) = 'P'; ++HL; M(HL) = 'R'; ++HL; M(HL) = 'N';
     DE = fcb1; C = search1; CALL bdos; AF = A OR A;
     LOOP
     IF NOT POS BREAK;
```

```
        ROTATE A LEFT 5; A = A + X'80'; L = A; H = 0; ++HL;
        A = M(fcb1); M(fcb2) = A;
DE = fcb2+1; C = 11;
     LOOP
     IF ZERO BREAK;
   A = M(HL); M(DE) = A;
   ++HL; ++DE; --C;
     REPEAT
A = C; ++C; C=21;
     LOOP
     IF ZERO BREAK;
   M(DE) = A;
   ++DE; --C;
     REPEAT
        DE = fcb2; C = openfl; CALL bdos;
IF A = A OR A NEG GOTO getnext;
        DE = fcb2; C = readseq; CALL bdos;
IF A = A OR A NOT ZERO GOTO getnext;
HL = defdma;
        LOOP
   A = M(HL); ++HL;
REPEAT WHILE A:formfeed NOT EQUAL;
        LOOP
   E = M(HL); C = outchr;
   PUSH HL; CALL bdos; HL = POP;
   A = M(HL); ++HL;
REPEAT WHILE A:newline NOT EQUAL;
        getnext:
DE = fcb2+12; A = 21; AF = A OR A; C = A; A = 0
   ;
     LOOP
     IF ZERO BREAK;
   M(DE) = A;
   ++DE; --C;
     REPEAT
A = M(fcb1); M(fcb2) = A;
DE = fcb2; C = search1; CALL bdos;
HL = fcb1; DE = fcb2; A = 9; AF = A OR A; C = A
   ;
     LOOP
     IF ZERO BREAK;
   A = M(HL); M(DE) = A;
   ++HL; ++DE; --C;
     REPEAT
DE = fcb2; C = searchn; CALL bdos; AF = A OR A;
     REPEAT
     GOTO warmboot;
     fcb1: RESERVE 33;
     fcb2: RESERVE 33;
     version MODULE NAME start;
     END PROGRAM start;
```

End Listing Three

OF INTEREST

by Michael Wiesenberg

Save $9395

Qwerty, Inc., thinks you are about to rush out and spend $10,000 and up on a 68000 development system for your Apple. They want to save you money with their $695 **Qpak-68** system which includes a plug-in board with 68008, the software compatible 8-bit version of the 68000, an editor/assembler, a debugger, and documentation. The processor is driven directly by the Apple's 7.16 MHz clock; so it runs in parallel with the 6502, directly using and sharing the Apple's memory and accessing the same peripherals. The board has local 8K memory in EPROM and 2K RAM and is expandable to 32 and 8.

Polish Apple

The Great Creator, from The Professor and Gessler Educational Software (who claim to be America's oldest and largest publisher and distributor of supplementary foreign language teaching materials), is a program for teachers to create multiple choice and fill-in-the-blank questions on any subject, including 17 foreign languages. Special alphabets are available for those languages needing accented letters and letters not found in English. The program can explain to the test taker why his/her answer is correct or not, with immediate feedback. It also keeps track of the user's score. You need Apple II+, IIe, or a compatible, with 48K and one disk drive, and $399.95 for the Great Creator; or $5.00 for a demonstration disk, price applicable toward the "real thing."

CP/M For Less

Interceptor Computer Corporation has Z80 CP/M cards for Apple and Franklin for $95 and 80-column cards for the same price.

Saving Sorcerers

Those with Sorcerer computers who feel they have unsupportable machines on their hands since Exidy sold off its Data Systems Division can take heart. **Electronic Installations** has everything you ever wanted to replace faulty parts in your computer and add peripherals you may never have thought possible. You want 80 columns for your monitor instead of the 64 standard? $395 gets you the board. How about a hard disk? $1550 for 27Mb and $4000 for 47. All the ROM PACs are available, including BASIC *1.1*. Electronic Installations has much software, all of it supported with a preset amount of free consultation. Need a new BRUCE PROM? $8. An 8086 S-100 board is in the works. Send for a catalog. Electronic Installations can also help you with Dynasty Computer's Smart Alec problems.

Z80 Disassembler/Reassembler

D80M, from Microtrain, disassembles Z80 code, and is completely compatible with Microsoft's Macro-80. Even where D80M cannot fully resolve all labels, it can reassemble any file it disassembled. $175. Microtrain also offers what, from the description, sounds like an excellent CP/M word processor for the price, **Scribewriter IX**. $75.

64 Things (and 20)

Abacus Software has **Zoom Pascal 64** for the Commodore 64, a package with editor, compiler, translator, manual, and example programs. It produces 6502 machine code instead of p-code. The disk costs $39.95. Priced at $42.95, **Chartpak-64** produces high resolution pie, bar, and line graphs, and saves them to disk and prints them on graphics printers. **Assembler/Monitor 64** has full screen editing of source programs, object code assembly to memory, disk, or tape. It also has a complete symbol table listing, file chaining, 6502 syntax, global search, code disassembly, block transfer and compare – all for $32.95. **Super Disk Utility** copies disks in four ways, sector by sector, files in use, files specified by user, and adding files to a single disk. It also displays directories and the contents of any disk sector, and permits updating the contents of any sector and rewriting back to disk. $22.95. **Graphics Designer 64** is a menu-driven editor that produces high and low resolution graphics for screen and printer, and displays in timed sequence. $34.95. **Pool-64** and **Pool-20** are high resolution nine ball and straight pool for the 64 and Vic 20. You control speed and angle of cue ball; and it even has sound effects. The press release on this one tells us that "*caskette*" costs $14.95 and "*dissette*" $17.95. (Yes, I know what they mean, but what a wonderful misprint!) **Datamat-64** is a $32.95 DBMS. **Tiny Forth** is for both 64 and 20 – $19.95 on cassette and $22.95 on diskette. (The 20 will need an 8K expander.) Who says you can't use a computer to manage your checkbook? **Checkbook Manager 64** lets you enter checks written, deposits, checks returned, and service charges, create statements, list all checks and deposits, and change or edit a check. It supports a printer, and has backup and recovery features. $22.95. Abacus also has a book, **The Anatomy of the Commodore 64**. It's 300 pages on machine and assembly language programming, music synthesizers, graphics, BASIC, a comparison of the 20 and 64, I/O, and ROM listings, for $19.95.

WP for 100

Scribe, from Chattanooga Systems Associates, is a word processor for Radio Shack's Model 100 that greatly extends the portable's somewhat limited TEXT entry program. $24.95 plus p. and h.

Great Effort Rewarded

An A (E?) for Effort (C for *chutzpah*?) goes to Renaissance Technology, whose press release for **Le Switch** came to us in a $12 Federal Express overnight letter. The product is a switch box and connectors for attaching two printers to one parallel port, $155, or serial, $125.

And Twice as Many

And if two ports aren't enough for you, the **Digital ESP-1 Port Expander**, from Digital Laboratories, creates four serial RS232C ports from one, with instant switching to each under software control, or with manual override. $395.

Talk to Your Typewriter

Do you have an electronic typewriter and a computer, and are cursing because now you have to buy a printer too? **Supercord**, from Cord Ltd., links nine different brands of electronic typewriters to 20 different computers, and they are adding more by the minute. Supercords are available for most

output ports, and Supercord II has a 4K buffer. I don't usually list a product for which the release gives no price, but this seems like a good one.

Wooden It Be Nice?

Don't like stark, utilitarian metal computer furniture? You can get a handsome solid wood computer table, 27 inches high, 38 inches wide, 24 inches deep, for $98 from **Casual Concepts.**

Sign Up for ACP!

The **Association of Computer Professionals** is now soliciting members to get involved in programming and program development, hardware manufacturing, teaching, consulting, writing, publishing, or any other activity related to computers. No qualifications are set regarding age or education, nor is distinction made between part- or full-time involvement, or salaried, self-employed, or free-lance practitioners. Charter membership for one year costs $45 for a regular membership, $20 for a student membership. After January 1, 1984, it'll be $55 and $25. Corporate, club, and educational memberships are also available. Membership includes the newsletter, *ACP NEW$*, assistance to members in marketing software and computer literature, grants and awards to members, posting in the newsletter of job opportunities, reduced prices for computer products, and conferences. ACP will send you a free newsletter and membership information.

Don't Pour Coffee on Your Diskettes

Now that you are a computer professional, how do you protect your investment? Secure your equipment, keep liquids away (spills are the major cause of accidental damage to computer equipment), back up programs and data, watch for fire hazards, plan your installation, use proper electrical outlets and wiring, and – I've said this before, but it should be repeated regularly – insure your equipment. Most homeowners' and renters' policies do not cover personal computers used even partially for business or claimed as tax deductions; and if they do, in the case of software, only the replacement cost of the media is covered. You have all your records and data on a Winchester which gets zapped by a power surge. Most policies would cover only replacement of the disk medium, and then only if it was physically damaged. But in a power surge, this might not even happen; most likely, all you would lose would be the data. **Safeware**, though, from the Columbia National General Agency (CNGA), a division of ARMCO, covers hardware and software, with protection against fire, theft, power surges, and accidental damage. A fifty dollar deductible coverage for a typical system with $5,000 worth of hardware and software is $60 a year.

Protect Your Software

Security, from The Answer in Computers, gives password protection to programs without affecting normal processing, and with no modifications to the operating system, for $50. **Security-Plus** adds protection to individual files ($100) while **Security-Plus-Log** adds a complete record of all program and file activity ($150). A demo disk costs $25, good toward the purchase of any package. You need CP/M-80 2.2, CP/M Plus, or MP/M-80.

Paris in the Springtime

Looking for an excuse to go to Europe next spring? Join an expected 35,000 others who will combine business and pleasure by attending Europe's oldest and largest microcomputer show – the **Ninth Micro Expo** at the Palais des Congres in Porte Maillot, Paris – produced by Sybex-Paris, May 22-26.

■■J

(Continued from page 9)

dates after 2079 are accurate, but this is not very important because at most only a 100-year period is usually needed. The only other difference in this listing is the small round-off parameter in lines 290, 320, and 330 which handles the differences between INTeger arithmetic.

I also loaded the same listing (the PRINT symbol was changed to suit) into CBASIC and MBASIC. CBASIC uses 14-digit arithmetic and has no problem with this program. MBASIC has a limited floating-point accuracy (7 digits in single precision), but MBASIC also has double precision which uses 16 digits (just DEFDBL A-Z at the beginning of the program). Single precision will give the same results as North Star 8-digit BASIC; double precision can use dates much later than North Star 14-digit BASIC. I tried using the MOD function in MBASIC, but could not get it to work with double precision (I always get an overflow error). Maybe someone who uses this BASIC more than I have can comment on how to use MOD in double precision.

CBASIC and MBASIC (and occasionally North Star BASIC) can be considered very common versions of the BASIC language. They certainly can handle Mr. King's code. I have not tried his assembly language routines.

On other topics, I have been getting more into CP/M recently and have gone through all of the *DDJs* from Volume 3 on. There are a lot of good programs in there and I really enjoy most of the Letters column. I have never subscribed to *DDJ* because of being inundated with esoteric languages. I have no use for Forth, Pascal, or C. BASIC suits me fine and can handle all of the programs I have been writing. I have been dabbling in assembly language, but do not care to learn it.

I guess that is all I have on my mind. Please do not forget us BASIC and assembly language fans!

Sincerely yours,
Saul G. Levy
2555 E. Irvington Rd. #47
Tucson, AZ 85714-1909

(Listing begins on page 15)

A Spelling Lesson for the Doctor

Dear Editor:

Proper spelling may seem trivial and irrelevant to you, but your November 1983 issue contained the word "kernal" plastered all over. This angered me, because sloppy spelling is the tip of an iceberg – an iceberg of contempt for the education and competence that many of us have worked hard to achieve.

S. Richard Mateosian, Ph.D.
Consultant, Comp. Systems
2919 Forest Avenue
Berkeley, CA 94705

What can we say? We received admonishment from several sources the same morning that we received the above letter. The word is commonly misspelled, so commonly that even our meticulous proofreader missed it – no doubt mistaking it for one of those coined computer terms. We will all, of course, be more careful in the future. Thanks to all those who keep us honest. At least we were consistent . . . – *Ed.*

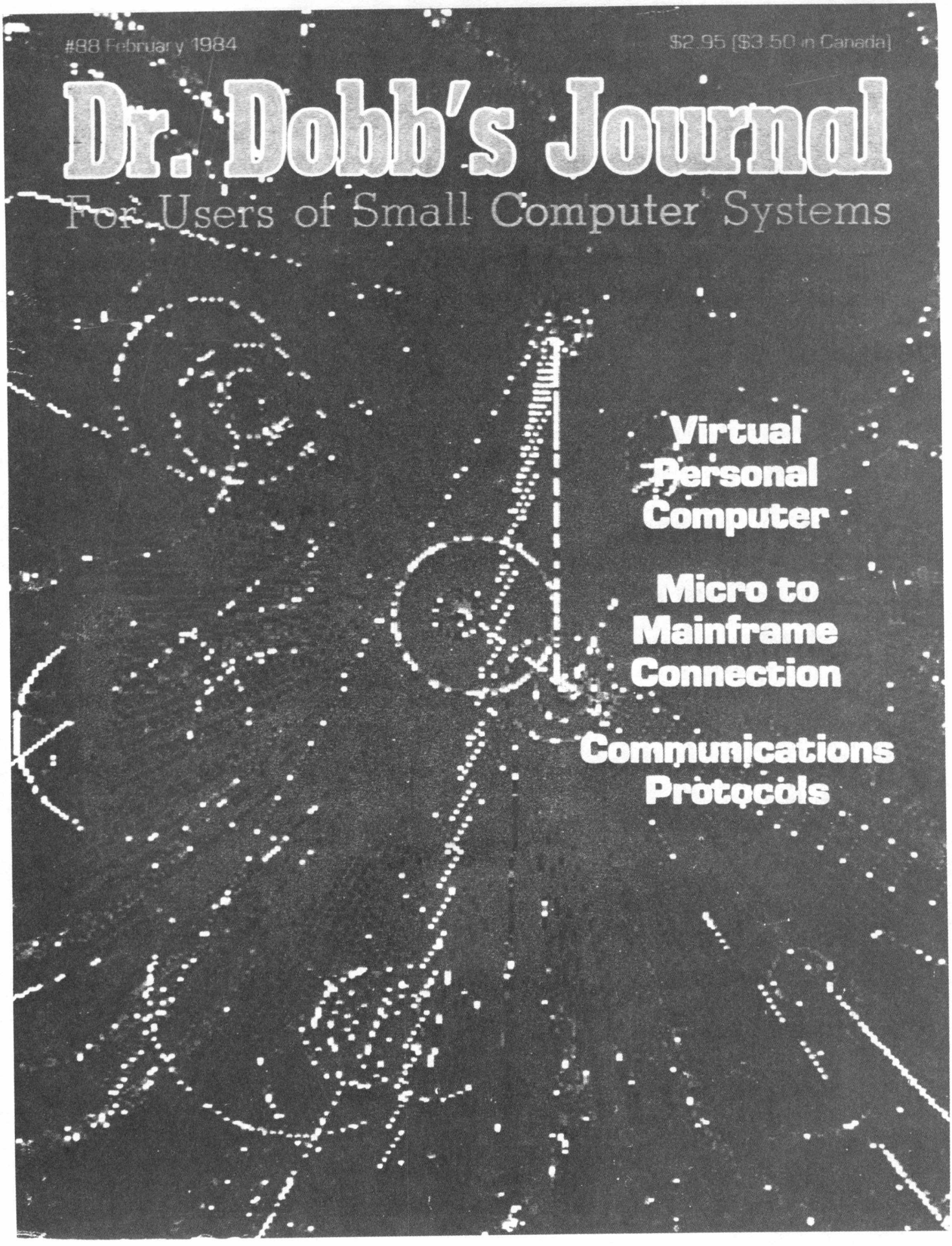
■■J

#88 February 1984

$2.95 ($3.50 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



Virtual Personal Computer

Micro to Mainframe Connection

Communications Protocols

Dr. Dobb's Journal
For Users of Small Computer Systems

February 1984
Volume 9, Issue 2

CONTENTS

ARTICLES

20 Micro to Mainframe Connection
by Keith Coye and Alvin Grossman

How does one decide what to use when linking micros and mainframes? Follow the evolution of one project through evaluation criteria, hardware and software selection, configurations, applications, and future direction.

26 Communications Protocols
by Leslie Brooks

"Communications is a very large, very deep swamp, and it contains many alligators. If you jump in with no preparation you will get eaten alive." Join the author as he examines a number of common communications protocols – what they can do and why or why not to use them, as well as a word on accuracy.

30 Unix to Unix Network Utilities
by Ron Coleman

This relatively unknown, yet powerful, set of network utilities is now available on most of today's Unix micro machines. Learn about their potentials, as well as their pitfalls.

32 VPC: A Virtual Personal Computer for Networks
by Jim Fleming

The VPC is a system for integrating timesharing and personal computer systems. The author describes this software-implemented system, including some discussion of design tradeoffs, as well as some illustrative code.

60 PABX and the Personal Computer
by Ted Rohling

The Private Area Branch Exchange is now a prime candidate for use as a local area network. Take a look at what a PABX is and whether or not it is the LAN of the future.

64 Basic Language Telecommunications Programming
by Robert S. Broughton

The author examines some aspects of how to upload files to systems that will not allow installation of programs such as MODEM7 and how to get BASIC to read the status of your microcomputer's hardware.

70 U.S. Robotics S-100 Card Modem
by Michael McKillip

The author describes some of the problems and tradeoffs that were involved in designing this 1200-baud S-100 modem, as well as a few of its features. It presents an interesting look into the modem board development process.

DEPARTMENTS

6 Editorial

7 Letters

10 Dr. Dobb's Clinic
System Languages; Keep Talking; Move Over, Ray

14 CP/M Exchange
Bringing up CP/M Plus: Implementation and User Report.

76 Software Reviews
Friday! SAL/80, Filebase, Oubliette

81 Book Reviews
C Programming Guide, Microcomputer Graphics Techniques and Applications, Introduction to Numerical Computation in Pascal, Automation of Reasoning

84 16-Bit Software Toolbox
PC-DOS Close Function, The Microsoft Assembler, Highly Recommended Software

94 C/Unix Programmer's Notebook
Reader feedback and further discussion of C layout standards

98 Of Interest

Publisher – Jane Nissen Laidley
Editorial
Editor – Reynold Wiggins
Managing Editor – Randy Sutherland
Contributing Editors –
Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg
Marketing
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Marketing Coordinator – Ed Gueble
Advertising
Advertising Director – Beatrice Blatteis
Advertising Sales –
Alice Hinton, Walter Andrzejewski
Circulation
Circulation Director – Terri Pond
Circulation Assistant – Billie Greenwood
Production
Art Director/Production Manager –
Barbara Ruzgerian
Production Coordinator –
Shelley Rae Doeden
Production Assistant – Alida Hinton
Typesetter – Paula Fairchild
Cover Design – Barbara Ruzgerian on Via Video System One
Special Thanks – Berkeley School of Computer Graphics, for technical assistance

Copyright © 1984 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; Euro Computer Shop, 182 Rue du Faubourg St. Denis, 75010, Paris, France; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014, Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian Distributors)*

EDITORIAL

This month we are pleased to present a special issue on telecommunications. We appreciate the response to the issue and regret that we could not include all of the material that was submitted. We will try to present other items on the subject in later issues. Here, however, we hope that you will find an interesting variety of topics.

As was noted briefly in the VPC article, the complete code listing was sufficiently large that we included only some of the illustrative routines. We are willing to publish the rest of the code, however, if there is sufficient interest. Those interested in seeing the remainder of the VPC level one system may drop the editors a note indicating this, or simply use the editorial response card included with the magazine.

* * *

Our referee program is now in place and working. As we explained in the initial call for volunteers, these referees are people who offer their thoughts and suggestions on material we are interested in publishing in *DDJ*. We are very please with the useful comments we are receiving, and we extend our thanks to all those who have offered their services.

This is the first issue in which we have really made use of the new referees. While we will publish a complete list of the board of referees next month, we would like to thank the following members for their timely insights on this issue:

Robert Blum,
Contributing Editor, DDJ
Patrick Burnstad, Lands End Computers
Keith Coye, Software Consulting Corp.
Mel Cruts, Triology
A. Gomez, Telecomp, Inc.
David Kirkland, Baker & Botts
Patrick Lynch,
Tandem Computers Inc.
Darryl E. Rubin, Rolm Corp.
Joseph Sharp, Micro Science Assoc.
Charles Wilde, IEEE, ACM

My thanks also to David Harris of PCC's PCNET Project for his assistance.

* * *

We have a number of useful topics planned for the near future. Many people have called looking for a runtime library for Small-C v2, presented in the December 1982 and January 1983 issues of *DDJ*. This spring we will publish the long-awaited library. In addition, we hope to have some other items available for use with the compiler. You will of course see more of our usual fare – languages, tools, algorithms, and so on – as well as a few surprises. Next month, we begin a two part article on a public-key data encryption system.

Talk to you next month.

Reynold Wiggins

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines and the current compensation schedule by sending a self-addressed, stamped envelope.

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94026. Second class postage paid at Menlo Park, California 94026 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94026 · 415/323-3111 **ISSN 0278-6508**

LETTERS

More Forth Writeability

Dear Dr. Dobb,

Before I begin, this is a *letter,* not an article – so please don't pay me yet! I received my first *DDJ* yesterday (whoopee, my own copy) and was not disappointed.

I was delighted that you decided to reprint Harvey Glass's article "Towards a More Writeable Forth Syntax" and thus make it available to us masses. There is a growing awareness, despite the jealous attempts at standardization by the purist elite, that Forth is simply not readable (or writeable) enough for broad acceptance. A group of languages is being developed in an attempt to make it more palatable and consistent. These include STOIC (J. M. Sachs, S. K. Burns, "STOIC – An Interactive Programming System for Dedicated Computing," *Software – Practice and Experience*, Vol. 13, 1-16, 1983), PISTOL (E. E. Bergmann, "PISTOL – A Forth-like Portably Implemented STack Oriented Language," *DDJ*, No. 76, Feb. 1983), and a modification of PISTOL which I am in the process of developing called REPTIL.

The most important departure of these languages from Forth is that a string is also a fundamental data type. Thus every name begins life as a lowly string and can be subsequently manipulated or converted to a more elite *verb* which "does" something by simply assigning it an action using the colon defining verb. In REPTIL, for example, a definition for SQUARED could be:

```
'SQUARED DOES: DUP * :END
```

This is a true "Reverse Polish" approach. Forth defining words have an inconsistent "infix" format, thus:

```
: SQUARED DUP * ;
```

The second major departure is that compilation is done on a line basis rather than compile and execute on a word-by-word basis. In some cases execution can be deferred over a few lines. The various structured constructs are thus allowed in both immediate and colon definition modes.

The syntax proposed by Harvey Glass is delightful – it considerably improves readability. I don't believe that Reverse Polish is less natural than Infix; in fact, in a recent article on Creole languages (D. Bickerton, "Creole Languages," *Scientific American*, July 1983) we see that postfix notation, "The poor people all potato eat," or prefix notation, "Work hard these people," is more "natural" than our formalized Infix heritage. What, then, is the source of Forth's lack of readability?

One reason is the cryptic symbolism, where nondescript characters like "@" "!" "," and "." take the place of more suggestive verbs *fetch, store, include,* and "=" (or *pop-and-display*). Another reason is that most humanoids prefer to associate their objects with names, rather than have them nebulously floating around on stacks. In Forth-type languages, however, all constants and variables are global, and their use thus considerably clutters up the structured modular nature of the programs. The concept of a local parameter, as well as the use of NOP's in the form of parentheses and commas, is a major breakthrough in this matter.

The use of parentheses has been recently proposed by Bergmann (E. E. Bergmann, "Languages & Parentheses" – *planned for a future issue of DDJ – Ed.*) for use in Forth-like languages, however not as simple NOP's. They are controlled by syntax checking in that execution is deferred until matching parentheses are established. The comma will surely upset the purists, as will the use of <DEFINE in place of ":". How can we throw away the cryptic ethnic symbolism of Forth with such disregard of tradition? Easily!

The REPTIL equivalent representation of the Ackermann function presented by Harvey Glass would be as shown in Figure 1 (at right). A few words of explanation are in order. All relational operators ask the question: Is it true or false? Thus they are all followed by a question mark, e.g., =0? – is TOS equal to zero? The wording of the conditional branch is designed for readability; I cannot accept the topsy-turvy Forth "IF – ELSE – THEN" hence they have been respectively replaced by "?THEN – ?ELSE – ?END."

The parentheses are those proposed by Bergmann and the comma is the NOP of Glass. STOIC originally proposed the vocabulary stack technique, in which, for instance, ASSEMBLER< will push the assembler vocabulary on the vocabulary stack and > pops it. Thus LOCAL> is a system vocabulary which does not have a permanent branch point. When invoked, it is linked to the latest *current* vocabulary.

:IS defines a constant (it *is* a value), and :HAS similarly defines a variable in the normal sense (it *has* a value, which can change); thus invoking I and J (defined as constants in the LOCAL< vocabulary) simply pushes their respective values to the stack.

Sincerely,
Issy Urieli
225 Highland Ave.
Athens, OH 45701

Curious about Ackermann

Dear Editor,

The function defined by Harvey Glass in the Nov. '83 issue is not the same "Ackermann function" as defined by Petersen in *Forth Dimensions,* III, 3, pp. 89-90. (Peterson's version is shown in Figure 2 at the right).

The fig-Forth code which Petersen gives to implement this function is garbled. Glass's standard Forth code is better,

```
'ACK DOES:
   LOCAL< 'I :IS   'J :IS

 I =0?  ?THEN              { J 1 + }
        ?ELSE  J =0?  ?THEN
                          { I 1 - , 1 } ACK
                      ?ELSE
                          { { J 1 - , { I , J 1 - } ACK } ACK }
                      ?END
        ?END   >
 :END
```

Figure 1.
REPTIL Equivalent Representation of the Ackermann Function

but still not correct. A correct implementation of the above function can be had by replacing the DUP on the second line of Glass's Figure 1, with an OVER.

I am curious about the origin of this function. Who is Ackermann, and where did he publish a definition of this function? Of what use is the function other than testing recursion in programming languages? Can any of your readers help with these questions?

Having asked the above, I imagine you may be curious to know how I can be so sure that Petersen's definition is "correct" and Glass's is "incorrect." The answer is that Glass's definition leads to infinite recursion with an unbounded increase in the depth of the recursion for all non-trivial values of the arguments while Petersen's leads to large, but finite recursion.

Sincerely,
Paul E. Condon
260 Devonshire Blvd.
San Carlos, CA 94070

```
ACK( I, J )  = J+1                    for I=0
             = ACK( I-1, 1 )          for J=0, I>0
             = ACK( I-1, ACK( I, J-1 ) )
                                      for J>0, I>0
```

Figure 2.
Petersen's Version of the Ackermann Function

Merci, Morrow

Dear Editor,

I purchased a Morrow Micro Decision from Priority One Electronics when they held their grand opening in Irvine. It is the first factory-assembled computer I have owned. For $2100, I got the computer with two double-density 5¼-inch drives (192K each), 64K RAM, WordStar, Basic, Microsoft BASIC, Correct-it, Logicalc, and Personal Pearl, the MDT 20 terminal, and COEX 80 printer.

The system worked wonderfully well for about a month, but then began occasionally to give the dreaded BDOS error on B. As it seemed to be getting worse, I loaded the system in my car, and drove to Irvine. At the store, I set it up, and of course everything worked perfectly. For about three hours I thrashed it on every piece of software it had failed on, to no avail. So I bought some new disks and a disk head cleaning kit and loaded it all back in the car.

Once home everything worked well again for almost two months. The key phrase is "almost two months." To be exact: 89 days from the date of purchase drive B quit. As I did not wish to drive from San Diego to Irvine again, I called and asked about warranty repair. Priority One in Irvine told me to call Morrow. I really wasn't expecting a cheerful reception from Morrow, but the lady I spoke with was extremely courteous and asked if I had contacted the dealer. I replied "yes," and then she said she would call me right back. About 20 minutes later she did and asked if I could mail the computer in. Of course I could, but I explained that I am in the Navy, and asked if she could have it back in less than three weeks since I was going overseas. Well, I know that things take time. I really didn't expect to have it back, but I wanted to have the computer overseas with me. Can you believe they would try?

To shorten this up, they did. And even a note inside telling me my diagnosis of the problem was correct. I am amazed. Of course you probably have heard good things about Morrow before, but I can't even get that kind of service on my $16,000 car. I wanted to make sure they got their kudos. The computer has been with me now for four months on a ship overseas, and that means vibration and infrequent use. The little jewel has held up marvelously – in fact, so well that I would seriously investigate nearly anything before hardware for a fault. I haven't even named this one yet, maybe Clark, for Clark Kent – you know, Superman in plain clothes.

In Navy talk, "Bravo Zulu" for a super job, Morrow.

ETC Howard L. Howell
USS Fletcher DD992
FPO San Francisco, CA 96665

A Plea for Xitan Help

Gentlemen:

I don't know how to tell you how much help you have been to me in my constant learning about computers. Even when I don't understand 90% of the magazine, I read through. Each month it gets easier.

Right now I could use a favor from one of your readers. I am the proud owner of a Xitan computer. That's right, Xitan. The trouble is the documentation needed for repair must still be the property of the first owner. I sure don't have it.

If anyone out there has the schematics, etc., for a Xitan, please let's get together.

Thank you for being there.

Yours very truly,
Larry Litwin
1935 La Habra Blvd.
La Habra, CA 90631

HELP for Readers

Dear Editor:

I wish to announce that the Small-C Help Facility, as published in the October 1983 issue, is now available from Pyramid Systems, Inc. of San Marino, California. Those who wish to use the package but don't relish typing it in by hand ought to appreciate the low cost nature of this source.

Thanks to those who have written concerning the package. I will be happy to keep *DDJ*'s readers informed of any known bugs and fixes.

Sincerely,
John Staneff
Route 1, Box 801
Ellensburg, WA 98926 ■■J

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

System Languages

We seem to have hit a community nerve with our blast at current compilers; some of you are actually writing. Great! Let's review somebody else's opinions for a change.

Most everybody agrees that we need a portable language for systems programming, in an implementation that produces efficient object code. Even Steve Newberry, who markets a structured assembler, likes the idea. He writes, "I suggest you take a look at Leor Zolman's BDS C (for the 8080) and Mark De Smet's C88 (for the 8086). They are so good that it is often hard to justify the use of structured assembly packages like my SAL/80. But there are, and I believe will long continue to be, situations in which . . . a high level language is simply not appropriate for the job. In those cases the structured assembly language provides the most economical solution."

Thanks for the tips, Steve. BDS C scores high in most published benchmarks and has an enthusiastic user group. However, it differs from the Unix standard at a number of points. The advantage of Aztec (and perhaps other C compilers) is its close match to Unix C. Standardization allows program portability, and that's supremely important in today's software market. Here's another point: most textbooks on C are based on the Unix standard. When a compiler deviates, it puts impediments in the way of the independent student. It's hard enough to learn a language on your own without having to translate the examples in the book.

Jim Howell of San Jose, CA, compiled our sort program under Introl C, using OS-9 on a 1 MHz SS-50 system. He reports a linked object file of about 10K bytes (versus our 16K) and execution times that were comparable (except for writing, which was only slightly slower than reading). He notes that our IGNORAD constant might well have been required because of a blank line at the end of the input file. Very possible.

David King of Woodside, CA, did an informal benchmark comparing Aztec C, Microsoft's BASCOM, and Digital Research's PL/I-80 and CBASIC. The test program leaned hard on floating point accuracy and looping. The fastest execution time came from PL/I-80 (10 seconds versus 15 for BASCOM, 16 for CBASIC, and 19 for Aztec C). PL/I and BASCOM produced 9K object files, versus Aztec's 11K. Compile times were comparable. Only PL/I and CBASIC were able to produce the correct answers to the limit of their precision.

King takes off from these results to reach some conclusions on the relative merits of the languages involved. This is something we all have to be careful about. It is impossible to benchmark a *language*; one always benchmarks an *implementation*. Every aspect of compiler performance – portability, execution time, object size, compile time, and numerical accuracy – is a product of the implementation. We can live with just about any Algol-descended language for systems programming, so long as its implementation adheres to a standard and produces efficient object programs.

David Clark of State College, PA, sent a bulky package of helps for Dwight Irving, who was struggling to implement the UCSD p-System. With them, Clark sent some helpful comments on Aztec C:

"The failure to link the 'write' function that you mentioned is not an error. The standard I/O library includes that function for system-level disk I/O. The library and the 'write' function are both described in K & R.

"The slowness associated with redirecting output to a disk file is a problem with the Aztec version of the library. Console I/O (stdin, stdout, stderr) is normally done one character at a time. When redirecting output to a disk, the I/O is still done a single character at a time. Since no buffering is done, a sector must be read or written for each character. [*Aha!*] Most CP/M implementations perform redirected I/O by saving the characters in a buffer that is some multiple of the sector size. When the buffer becomes full or the file is closed, the entire buffer is written at once.

"The size of the code file is due to the way the library is implemented. When the library is created, all of the functions present in one file will be compiled into a single module. At link time, the library is scanned for needed functions. When one is found, its entire module is linked. I have two other versions of C that construct the library by compiling just one function at a time. They can then link a single function. Of course, building a library can be pretty tedious when you have dozens of functions to change. [*That's why God made submit files, David.*]

"There is a public domain rework of the Aztec library that remedies these two defects. Redirected I/O has been cleaned up and is much faster. The library files have been broken into much smaller pieces, too. The library is mostly the work of Herb Shulz and can be obtained from a friend of a friend of a friend. . . ."

Jack Purdum of Indianapolis, IN, has another perspective on the size of a C run-time library:

"I'm not unbiased on C compilers (we market one that competes with Aztec). However, all compilers suffer similar limitations in regards to a number of your comments. First, printf(), scanf(), and fprintf() are *very* large functions because they are required to do so many things. Your program uses only a fraction of the power of these functions. This 'H-bomb-to-kill-an-ant' problem comes with using built-in functions from the library. If code size is important [*when the heck is it not?*] writing a new function that does only part of what the complete function does is one solution – your xprint() is an example. We (Ecosoft) have compiler switches that allow an integer printf() to be used, thus avoiding linking the floating point routines. Another switch avoids the file I/O options when doing just screen I/O. While this helps, it's not a total solution."

You make an important point, Jack. The scanf() and printf() functions are defined to convert between *any* data type and ASCII. Furthermore, the conversion string argument that guides them may be a variable; ergo, the compiler can't know at compile time which conversions are actually required. As a result, a single call of

```
printf("hello")
```

may cause the linking of the library routines for conversion of signed and unsigned integers, longs, floats, you name it. And they link other support routines, so any printf() call will cascade just about the whole run-time library into your program.

We can think of two solutions. First, instead of your compiler switches, why don't you put the printf() and scanf() functions in a separate library, and distribute two versions of that library? One version would support only strings and short integers; the other would be full-function. Name the one you want in the linker command line. Use macro facilities cleverly enough, and the two versions of

the functions could be compiled from the same source file.

Second, why do the standard functions have to be written in C? Sure, Drs. K & R showed model support functions in *The C Programming Language*, but aren't they to be taken as just that, models? Use them to get your compiler off the ground; use them in Beta-test; use them as executable functional specifications. But when it comes to the final version of the most heavily used code in every user program, shouldn't you hand-code them in the target assembly language? Isn't it the whole point of a high level language that one person, the compiler designer, takes on the burden of using the machine effectively, thus lifting that burden from the shoulders of many people? A hand-tuned assembly version of printf() ought to run rings around a compiled version, and the payoff, multiplied over thousands of user programs, more than justifies the effort.

Keep Talking

This discussion is based on our assertions that a portable, efficient, systems programming language is desperately needed, and that no existing microcomputer compiler is adequate to meet the need. We'd make a third claim: compilers are as poor as they are only because we users have not demanded better ones.

If you disagree with any of these claims, feel free to write. Write too if you have counter-examples: compilers that do produce efficient code, cases where assembly language is needed regardless, or proof that certain language features can't be made efficient no matter how much demand there is. Also, does anybody know who did the first optimizing compiler? It was a Fortran for the IBM 360 done in the late 60's, but we've lost the reference. What other optimizers have you used, and how well did they work?

Move Over, Ray

Last month we tromped all over Bob Blum's columnistic territory and promised, in a fit of naked ambition, to move in on Ray Duncan's this time (we columnists have strong territorial instincts). All this comes about because what goes into this column is – in the absence of reader input, hint hint – whatever happens to have passed over the columnist's desk or through his keyboard in the weeks preceding the deadline.

What was passing through our keyboard in recent weeks was the final draft of a book on Concurrent CP/M, in support of which Texas Instruments had very generously loaned a Professional Computer with CCP/M installed. The Professional Computer, or "Pegasus" as we insiders call it, worked very well, and its CCP/M did, too. This version of the system uses the "3.1 BDOS," the same level of file system as used in CP/M Plus. (You can tell the 3.1 BDOS by its use of the INITDIR command to set up disks for timestamping.) Its BIOS and system integration had been breathed on by DRI, which may explain why we found no bugs or performance problems in it.

There was a command missing, however. CCP/M does not implement the "file search path" notion of CP/M Plus. There is only a single "system drive" which will be searched when a command can't be found on the default drive.

There are various ways of choosing the system drive. The default choice is set when the system is generated. In at least the first edition of CCP/M for the IBM PC, the system drive is set automatically as the highest-lettered drive in the system. The generic version of CCP/M includes a command, SYSDISK, that will change the system drive. In the TI version of CCP/M, the system disk is either diskette drive A: or, if you have one, the hard disk on drive E:. The SYSDISK command was omitted; there is no way of changing the system disk.

There are times when you might want to change the system drive. If you have only diskette drives, the system disk should at least be different from A:, the usual default drive. A: will be searched anyway, so the second search is wasted unless the system disk is another drive. Again, if you have an electronic pseudo-drive, you might want to make it the system drive after you have brought the system up and initialize it. We thought that the lack of a SYSDISK was an unnecessary handicap on the TI system.

With a little research, we figured out how to change the system drive. A short assembly program does it by putting the drive number in the System Data Area (see Listing One, below). As a bonus, you can change four lines to create another program which will change the temporary drive. The temporary drive is the drive on which the system will build scratch files, especially the VOUT files of data from buffered consoles. ■■J

Clinic Listing (Text begins on page 10)

```
;=============================================================
;
;       SYSDRIVE  [ d: ]
;
; SYSDRIVE is a simple, 8080-model, program that reports on and
; sets the system drive-letter under Concurrent CP/M.  Some
; CCP/M systems are delivered with a SYSDISK command that does
; the same job.
;
; Given no command argument, SYSDRIVE merely reports which drive
; is the system drive.  Given a drive-letter and colon, it will
; set that drive as the system drive.
;         THE DRIVE-LETTER IS NOT VALIDATED.
;
; The same source file becomes a TMPDRIVE command by changing
; the indicated ((operands)).
;
;=============================================================
```

Clinic Listing (Text begins on page 10)

```
;
Bdos            equ     224 ; interrupt number for BDOS
C_WriteStr      equ     9   ; code to display a string
S_Sysdat        equ     154 ; code to get System Data Area
Sysdisk         equ     4Bh ; offset of Sysdisk in SDA
Tmpdisk         equ     50h ; offset of Tmpdisk in SDA
;
                CSEG    ; everything in the code segment
                ORG     005Ch
Operand_drive   rb      1 ; First-operand drivecode
                ORG     0100h ; code begins here
;
                mov     cl,S_Sysdat
                int     BDOS    ; BS:EX -> SDA
                mov     al,ES:Sysdisk[BX] ; ((ES:Tmpdisk[BX]))
;
                push    bx      ; save regs while..
                push    es      ; ..displaying msg
                add     al,'A'  ; make drive printable
                mov     msg1_drive,al ; and put in msg
                mov     cl,C_WriteStr
                mov     dx,offset msg1
                int     BDOS    ; display msg1
                pop     es      ; recover saved
                pop     bx      ; ..registers
;
                mov     al,Operand_drive
                or      al,al   ; was a drivecode given?
                jz      exit    ; (if not, stop)
;
                dec     al      ; 00=A, 01=B, etc
                mov     ES:Sysdisk[BX],al ; ((ES:Tmpdisk[BX]))
                add     al,'A'  ; make new drive printable
                mov     msg2_drive,al ; and put in msg
                mov     cl,C_WriteStr
                mov     dx,offset msg2
                int     BDOS    ; ..and print it
;
exit:           retf    ; far-return ends program
;
msg1            db      'System' ; (('Temporary'))
                db      ' drive is '
msg1_drive      rb      1       ; filled in from SDA
                db      ':',13,10,'$'
;
msg2            db      'System' ; (('Temporary'))
                db      ' drive set to '
msg2_drive      rb      1       ; filled in from operand
                db      ':',13,10,'$'
;
                end
```

End Listing

CP/M EXCHANGE

by Robert Blum

Over the last several months I have spent a little time talking about CP/M Plus and its new features. All of these discussions have been from the outside looking in, probing no deeper than just below the surface. Even from that point of view I was very excited about what CP/M Plus was built to do and its many new performance options.

CP/M Plus is now in daily use on my machine and is largely responsible for turning the tide against V2.2, which has been retired to second string and bench warming. As is the case with most new products, there are a few rough edges that need to be smoothed out, but nothing more serious than cosmetic defects. I still have a lengthy wish list of features that I would like to have. However, my list now contains desirable additions, not items that are necessary to plug holes in an inadequate system.

I will be talking more about CP/M Plus regularly in future columns. Some of the areas that I have already covered will be reviewed in more detail and, where appropriate, source code will be made available.

I won't be getting into a lot of detail this month. I want to first share a few of my experiences in bringing CP/M Plus up for the first time and my initial reaction to using it.

Implementation

I chose to rewrite my BIOS for CP/M Plus (V3.0) from the ground up rather than attempt to adapt my V2.2 BIOS code. This was the most desirable approach for me because of a few specific changes that I had made to enhance V2.2 which would serve no useful purpose under V3.0 and could potentially cause unnecessary problems. Starting out with a clean slate also allowed me to implement many of my wish list items without being concerned about their effect on other sections of an already working system.

Before starting any work I decided to write a complete *banked* BIOS that could be used as *nonbanked* simply by changing one IF statement in each program module. I also wanted to keep all related device-dependent logic in separate modules for ease of debugging. The sample routines provided by Digital Research follow this same scheme and were very beneficial not only as a guideline, but also as a basis to work from.

Since I am an advocate of Zilog mnemonics, my first step was to convert the sample routines from Intel mnemonics to the Zilog standard with XLATE2. Each module was then edited for aesthetics. This step is not necessary if you don't care for Zilog mnemonics or if you want to use the sample routines as written. There is no real benefit to using one method over the other. It's simply a matter of personal choice. No matter how you go about putting your system together, I do strongly suggest that you at least use the sample routines as a guideline. Otherwise, you will waste a lot of valuable time in unnecessary coding and debugging.

I was then ready to begin the real task of writing, or rewriting in some cases, all the routines to interface V3.0 to my hardware configuration. I was surprised when I finished after only three days because I had originally planned on spending about a week writing code. I think a week is a more realistic time estimate unless you are able to do as I did and shut yourself away to avoid any disruptions.

I was fortunate in being able to salvage a major portion of my V2.2 disk controller code which accounts for at least half of my complete system. Adding the Extended Disk Parameter Blocks and related control structures was an easy task that required very little new coding. The time-consuming parts were the new modules: character I/O module (CHARIO), boot routine (BOOT), bank selection and memory-to-memory move routine (MOVE). To complete my suite of modules I used DR's main module, BIOSKRNL, almost verbatim. Everything fell into place quite easily and looked like it should work without a considerable debugging effort. I wouldn't realize how complex the placement of code between common and banked memory would be until I started testing in a banked environment. But this is another subject altogether that I will talk about a little later.

It was my intention from the beginning to limit the complexity of my first test system as much as possible in hopes that my debugging effort would also be simplified. When generating my first nonbanked system with GENCPM, I answered "no" to as many questions as I could and limited buffer allocations to one each for both data and directory. Fortunately, luck was with me and on my third attempt I had CP/M Plus operational.

After running a few test programs I was confident enough to begin generating more test systems with a few more features enabled than the one before it. Everything was holding together better than I thought it would until I started testing my first *banked* system.

I was sure that my bank switching hardware was stable because I had written a diagnostic program to verify its integrity and had been using it daily with V2.2 without any problems. Using SID to assist in debugging a banked system is no help at all because once a bank is switched out SID will typically crash or cause its own share of strange errors. What is needed is a program that does not depend on the lower map of memory or any BDOS functions to operate. As it turns out, repeatedly changing the code, relink-editing and generating, and testing were practically the only way I had to get my banked version up.

Even though I had a banked version operational on the first day, it was prone to sporadic failures. Finally, after two weeks and the eradication of a few bugs, I was completely operational with a stable system. During this time I learned a lot about how CP/M Plus works by reading and rereading the manuals and just poking around in an attempt to find out what was going wrong.

Based on my experience, I cannot overly stress the importance of studying the manuals that come with the CP/M Plus system before charging off to bring it up. Even though the manuals have been drastically improved, a number of very important tricks are well hidden in the text. In addition, CP/M Plus is a completely new system with very little resemblance to its predecessor.

User Report

Using CP/M Plus is a joy. The extended command editing feature available on banked systems saves a nontypist, like myself, a lot of time. If a keying error is made in the middle of a command line it is no longer necessary to backspace and erase to the incorrect character and then retype the remainder of the line. With CP/M Plus you simply type CTL-A to nondestructively backspace over each character in the command until you reach the error. You then have the choice of using either CTL-G to delete one or more characters at the cursor position or of making insertions by typing the new text. If the command line is lengthy you can also skip to the right or left end of the string with a single keystroke.

The entire command line can also be redisplayed even after return has been hit and a transient program has executed. One of my most common errors is to misspell file names which causes difficulties at the retrieval stage. Provided no other keyboard entries have been made, hitting *CTL-W will* redisplay the last command line entered and allow further editing. This one feature alone has saved me from many frustrating moments at the keyboard, especially when I am in a hurry. One other very practical use for this facility is when the same program must be run against a number of files. Rather than setting up a submit procedure you can now easily edit the same command line as many times as necessary.

And last but not least, multiple commands can be entered on one line by separating them with an exclamation mark.

I couldn't be happier with the keyboard editing features except for one gaping hole. There is no way to redefine the control sequences assigned to individual functions. I would have thought that this medieval practice, akin to the Chinese water torture, would have been outlawed in a new development effort.

Submit file processing is as automatic as you want it to be. After proper specification of the search path with the SETDEF program, a search is made for the .COM file specified in the command line. If it isn't found the search is automatically continued, only this time a file type of .SUB is used. When the desired file is found, SUBMIT.COM is loaded for execution and the specified file is used as input.

Program input can be imbedded in the submit file by preceding each input line with a less-than sign. In the case of an errant program that tries to read program data after the file is exhausted, further input requests are directed to the keyboard. In addition, system commands are protected from being read as program data.

Limited conditional submit file processing can be done by prefixing system commands with a colon. This indicator tells the CCP to check the system error code for a nonzero value, and if true, skip the command.

And in my estimation the most useful new feature is nested submit files. You are no longer prohibited from calling submit procedures from other submit procedures. This is especially useful when doing any kind of batch processing.

Supercharged

Quite an uproar was heard when the truth was known about the LRU buffering logic of CP/M Plus. The most heated public discussion took place in *DDJ* a few months ago. Dave Cortesi took the time to uncover the LRU buffering logic, only to find that for sequential files, there wasn't any. The fact is, LRU buffering is disabled for sequential file processing.

Fortunately, Digital Research is not a company to be kicked for very long before rectifying a situation. The application note at the end of this column changes the LRU buffering logic to give equal priority to buffer usage regardless of how a file is accessed. Exactly what this change means in terms of program runtime is unknown as yet because I haven't run any benchmarks to make a comparison between a before and after system, although from outward appearance I would think that a substantial savings is in order.

Tips

When coding the physical device names in the character table, make sure they are entered in upper case. The DEVICE transient program converts all keyboard input to upper case before searching for a match in the character table.

Application Note

CP/M Plus V3.0, Patch 13, 5/1/83 BDOS Patch 02

Copyright © 1983 by Digital Research Inc.
CP/M Plus and SID are trademarks of Digital Research Inc. Compiled September 1983. Printed with permission of Digital Research (but see the Editor's Note in the listing).

Products and Serial Numbers that Require Updating: CP/M Plus V3.0, serial numbers 2-000-00001 through 2-000-XXXXX

Program: RESBDOS3.SPR, BNKBDOS3.SPR, BDOS3.SPR

Error Description:

These patches do the following:

- clear the multiple command buffer if CTRL-C is encountered.
- change the LRU algorithm that manages data DCBs (banked systems only).
- correct the problem that occurs if a BIOS READ ERROR is encountered during login on a permanent drive.
- correct errors that occur if directory write operations are performed to disks set to Read Only (R/O).
- correct random record I/O error that occurs when the random record number is greater than 3F000h.

Patch Procedures:

Make a backup copy of RESBDOS3.SPR, BNKBDOS3.SPR, and BDOS3.SPR before making any changes. The program SID is required to make the changes. The changes are made by the sequence of commands in the Listing (below). ■■J

CP/M Exchange (Text begins on page 14)

```
15C>REN RESBDOS3.SAV=RESBDOS3.SPR
15C>SID RESBDOS3.SAV
C:SID      COM
CP/M 3 SID - Version 3.0
NEXT MSZE  PC  END
0900 0900 0100 CCFF
#S41A
041A F8 FC
041B C8 .
#S797
0797 03 07
0798 06 .
#WRESBDOS3.SPR
0010h record(s) written.
#G0

15C>REN BNKBDOS3.SAV=BNKBDOS3.SPR
15C>SID BNKBDOS3.SAV
C:SID      COM
CP/M 3 SID - Version 3.0
NEXT MSZE  PC  END
3600 3600 0100 CCFF
#A529
0529  CALL 2D24
052C  .
#S6C6
06C6 08 10
06C7 C2 .
#SB92
0B92 A0 97
0B93 09 .
#S13B1
13B1 C0 00
13B2 CD .
#A15ED
15ED  CALL 0EEB
15F0  JNZ 13FF
15F3  LHLD 2871
15F6  CMP M
15F7  NOP
15F8  NOP
15F9  JZ 13FF
15FC  JMP 2D40
15FF  CALL 1252
1602  CALL 1258
1605  .
#A1640
1640  JZ 2D6A
1643  .
#SW19DB
19DB 14F0 2D83
19DD CDC8 .
#A1B16
1B16  CALL 2D76
1B19  .
#SW1BA4
1BA4 14F0 2D83
1BA6 B9C2 .
#SW24F6
24F6 1E90 2D7D
24F8 36C3 .
#SW25F0
25F0 1E90 2D7D
25F2 52C3 .
#SW261B
261B 1E90 2D7D
261D A4CD .
#SW2704
2704 2510 2D3A
2706 20CD .
#A2C92
2C92  CALL 2D30
2C95  .
#A2CE5
2CE5  LDA 2D39
2CE8  ORA A
2CE9  NOP
2CEA  .
#S2D05
2D05 F8 F6
2D06 28 .
#S2D0D
2D0D F6 F8
2D0E 28 .
#A2F24
2F24  LXI H,0
2F27  SHLD FBBA
2F2A  SHLD FBB1
2F2D  DCX H
2F2E  DCX H
2F2F  RET
2F30  SHLD 28F4
2F33  SUI 3
2F35  STA 2D39
2F38  RET
2F39  NOP
2F3A  CALL 2D43
2F3D  JMP 2513
2F40  CALL 1377
2F43  LHLD 287B
2F46  MOV A,L
2F47  ANA H
2F48  INR A
2F49  RZ
2F4A  MOV E,M
2F4B  INX H
2F4C  MOV D,M
2F4D  MOV A,D
```

EDITOR'S NOTE:

This is still a preliminary version of patch 13, not *an official patch. It may contain some errors (though we have not heard of any yet). It is thus not endorsed for application by Digital Research. It is currently under review at DRI, and we had expected it to be approved by the time we went to press. Unfortunately, word of last-minute delays came too close to press time for us to withhold the listing. Thus we include this caveat. DRI will keep us updated, and we will let you know of its disposition and any changes that might be made.*

```
2F4E  ORA E
2F4F  RZ
2F50  LXI H,28AA
2F53  LDAX D
2F54  CMP M
2F55  JNZ 2D63
2F58  LXI H,4
2F5B  DAD D
2F5C  MVI A,FF
2F5E  CMP M
2F5F  JNZ 2D63
2F62  STAX D
2F63  LXI H,D
2F66  DAD D
2F67  JMP 2D4A
2F6A  CALL 1377
2F6D  LHLD 2871
2F70  MOV A,M
2F71  ORA A
2F72  RNZ
2F73  MVI M,2
2F75  RET
2F76  CALL 1139
2F79  LXI H,FD17
2F7C  RET
2F7D  CALL 1E90
2F80  JMP 1139
2F83  CALL 1162
2F86  JMP 14F0
2F89  .
#S3065
3065 82 92
3066 40 .
#S327F
327F 90 12
3280 49 .
#S35A5
35A5 00 48
35A6 00 21
35A7 00 09
35A8 00 24
35A9 00 00
35AA 00 21
35AB 00 00
35AC 00 40
35AD 00 49
35AE 00 00
35AF 00 91
35B0 00 24
35B1 00 80
35B2 00 .
#WBNKBDOS3.SPR
006Ah record(s) written.
#G0
```

```
15C>REN BDOS3.SAV=BDOS3.
    SPR
15C>SID BDOS3.SAV
C:SID      COM
CP/M 3 SID - Version 3.0
NEXT MSZE  PC  END
2780 2780 0100 CCFF
#A04C2
04C2  CALL 1E25
04C5  .
#S642
0642 08 10
0643 C2 .
#S846
0846 54 4B
0847 06 .
#SD6F
0D6F C0 00
0D70 CD .
#SW12A2
12A2 0E86 1E3E
12A4 CDC8 .
#A13DD
13DD  CALL 1E31
13E0  .
#SW1463
1463 0E86 1E3E
1465 78C2 .
#SW1B29
1B29 16A5 1E38
1B2B 63C3 .
#SW1B8A
1B8A 16A5 1E38
1B8C 3AC3 .
#SW1BB5
1BB5 16A5 1E38
1BB7 71CD .
#A2025
2025  LXI H,0
2028  SHLD 1EBA
202B  SHLD 1EB1
202E  DCX H
202F  DCX H
2030  RET
2031  CALL 0AF7
2034  LXI H,1CDF
2037  RET
2038  CALL 16A5
203B  JMP 0AF7
203E  CALL 0B20
2041  JMP 0E86
2044  .
#S2097
2097 02 06
2098 06 .
#S233F
233F 41 49
2340 20 .
#S26AC
26AC 00 24
26AD 00 12
26AE 00 24
26AF 00 90
26B0 00 .
#WBDOS3.SPR
004Dh record(s) written.
#G0
```

End Listing

Micro to Mainframe Connection

by Keith Coye and Alvin Grossman

Keith A. Coye, Consulting Services Corporation, 7214 Via Maria, San Jose, CA 95139.

Alvin Grossman, San Mateo Co. Office of Education, 333 Main Street, Redwood City, CA 94073.

Artwork: Joe Murray.

Long ago and far away there lived in the mind of Man a single large machine that could be all things to all people: the Main Frame Computer. To further Man's purposes the Main Frame Computer grew larger and larger and more complex. To tend its every need a priesthood of Programmers and Analysts evolved. They and only they were permitted to talk to the large machine. As in all priesthoods and cults, they spoke in strange tongues that only the large machine understood. All others had great difficulty communicating with the large machine and had to rely on the priesthood so that the machine would be bountiful and smile upon them and do their work.

Strangely, the people also found it difficult to talk to the priests, for the priests were more interested in talking to the large machine than to the people: a perplexing problem and one that did not seem to be amenable to an easy solution. Then into this inhospitable land of Babel rode a knight on a pure white charger: the Microcomputer!

All at once the people had within their grasp the world the priesthood had been promising, lo, these many years. They no longer needed the priesthood. Temples were smashed, large machines were shunned, a new day had arrived. All the people needed was access to a microcomputer and the proper software. Rejoice, the people were no longer beholden to others; they were in control of their own destinies.

Were it all so easy. Alas, the lack of software, the emphasis on Games, the difficult-to-understand manuals left the people wringing their hands. Perhaps they needed the priesthood after all. But in spite of the many early woes, software got better, people began to talk in fewer tongues, and the microcomputers got bigger and bigger. The Programmers soon found that they needed a means of exchanging information among the various systems, but each machine spoke in its own tongue. What was needed was a common language for transferring data among the various machines.

Here we leave our fairy tale and return to reality, although we very much hope that, like a fairy tale, there will be a happy ending. Many of us felt that the microcomputer was not a threat to our "empires" but instead was a godsend that would extend the boundaries of computing and hasten the arrival of true distributed processing. Although we were

"IN THE BEGINNING"

called foolhardy by many of our contemporaries in the priesthood, it was our perception that the microcomputer would usher in the day of popular computing and would effectively allow a melding of computing power and communications. It would put computing power into the hands of the users and, with it, the decisions on how they would want to use it.

Planning the Connection

At the San Mateo County Office of Education in the heart of the Silicon Valley, we began to plan our efforts way back in 1980 (it seems like eons ago). At that time our Three Year Master Plan stated:

"We have taken the view that micros are not antithetical to our current environment. In fact, through the development of this plan, the use of a variety of computers will result in an environment that will be more user oriented and make a full spectrum of computational power available to all users. We will move from being an organization with primarily an administrative focus to one that will provide the necessary conduits for delivery of a wide variety of systems.

"In addressing this new phenomenon it must be remembered that this is a new technology and not simply a cheaper or necessarily better way of doing things. What in fact is happening is that the nature of user interaction with computing has changed and the range of computing applications has expanded dramatically. The incorporation of micros into our 'systems' is especially fortuitous at this time. It will allow us to be able to offload minor and special-purpose functions and to expand our service offerings. It will also prolong the life of the current large computer (DECsystem-10). In addition to what has been traditional computer center functions, an expanded role will be added, that of more far flung telecommunications to expand the availability of computing to all districts and schools in our service area and to make their access easier.

"This plan proposes a means whereby we will become essentially a network, from the smallest micro to the major mainframe at our headquarters, with all users accessing and being able to obtain any files they want and need in a manner most advantageous to them."

Our goals at that time were explicitly stated as follows:

1. Move toward a marriage of micros/mainframes
2. Design an interface between the two machines to facilitate the easy communication and downline loading of data despite differences in operating systems, disk formats, and text format conventions

3. Design an easy-to-use system to allow "packages" of data to be downline loaded and then manipulated by a micro data base manager
4. Create a resource group to act as a primary "help" group to teach users how best to access and use their data bases on the mainframe
5. Design approaches that would allow the mainframe to be a primary storage and switching network and allow a great deal of the computing to be at the local level on micros

Initial Prototyping

We decided to prepare a prototype implementation connecting the DECsystem-10 at the San Mateo Center with a number of Radio Shack Model IIIs. They provided a large population of systems and were Z80 based. The choice of the Radio Shack Model III was based on the large population of this type of system in our area and the ease with which we could extend our software to other similar 8080-based systems. We felt that this would give us a good perspective on the challenges of connecting a network of non-CP/M systems to the DECsystem-10.

One of the first hurdles we faced was the choice of programming language for this application. We had both microcomputer- and mainframe-level coding to do and wanted to consider both ease of implementation and ease of maintenance. We evaluated different languages as follows:

1. DECsystem-10 assembly language – a good candidate from the perspective of its close relationship with the DECsystem-10 operating system and the ease of communication with operating system internals, but a liability since we could not transport the code to another mainframe or microcomputer
2. Fortran – a good candidate since Fortran runs on most systems, both mainframe and micro, but the liabilities included a lack of good string handling and isolation from the critical terminal and monitor internals required for this type of application
3. COBOL – again, a good candidate since COBOL runs on most systems, although it is weak in the area of efficiency and in the ability to control modem and communications areas

Further investigation led us to evaluate an emerging language, C, as a candidate. Lo and behold, it had the structure of a very high level language, nearly the efficiency of assembly code, and best of all was reputed to be highly portable. Off we went to choose a good C compiler

Evaluation of C Compilers

Very few C compilers were available three years ago, so our field of choice was rather limited. Most of the C compilers for the DECsystem-10 mainframe were implemented by universities or schools. The manufacturer had no product, so we turned to the educational area for help.

Several versions of C were located at the University of Utah and Tufts University in Boston. The version at the University of Utah was in development and had licensing restrictions that prohibited its use as a future commercial product. The version created by David Krumme in the Mathematics Department at Tufts was a good compiler although very newly developed.

On the microcomputer side, the BDS C compiler was one of the few that had any proven track record for the microcomputer community. Despite its nonstandard library and the difficulty of linking in assembly code, we chose it as our initial compiler.

We quickly discovered that the non-CP/M environment required a special library and support for the TRSDOS operating environment. Our strategy was to compile the main application code on a Z80 development machine under CP/M, unload the "com" file into an Intel hex file with an origin suitable for the TRS80, transfer the hex file to the TRS80, and create a special loader on the TRS80 to create an executable file on the TRS80. Needless to say, this was a complex, time-consuming, and often frustrating process. The debugging time was astronomical! Each bug that was discovered required a complete recompilation, unloading, transfer to the TRS80, and reloading to try again.

It soon became obvious that the process of transporting the code from one micro environment to another had to be improved. We decided that we must find a C compiler that ran on more than Z80-based systems, had a standard Unix-like library, and allowed easy integration of assembly code using standard linking loaders. The C code needed to be transportable to other systems with an absolute minimum of source code change.

The Final Selection

Enter Aztec C from Manx Software!

Early testing showed that, although the compilation and execution speed was slightly slower than the BDS C compiler, the claims for portability, assembly code integration, and standard library functions were true. We immediately converted the BDS C source code to the more standard Unix-like syntax used by Aztec C.

The communication software, now called XPRESS, came to life, and we found that the code written in Aztec C was portable not only to other microcomputers but also to the DECsystem-10 mainframe. Our strategy was to create common code segments for all machines and to design "personality modules" to adapt the specific operating system, modem, and communication requirements. The TRS80 version was a success and soon requests came in for Televideo, Apple, CompuPro, Radio Shack Model 2, and Radio Shack Model 16.

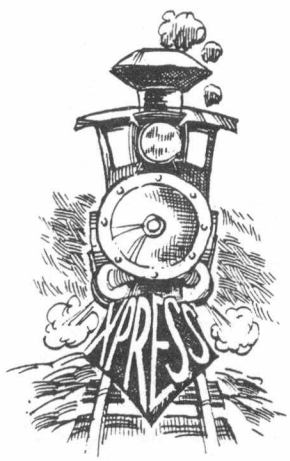
Then the IBM PC was announced and the race was on to get the C compiler up and running on the PC. Manx had not yet released its version for the PC so an alternative vendor, Computer Innovations, was selected as an interim candidate. The compiler was good in all respects, but it did require some modification since it did not adhere to some of the I/O syntax conventions used by Aztec C. Shortly thereafter, Manx released its Aztec C for the PC and we decided to continue with it for the PC development.

Software Design

The internal software design of XPRESS is described below. It may be of interest to those readers who enjoy the more technical side of the communications arena.

XPRESS is designed to allow a common memory area to be used as required for terminal communications, file transfer, and general-purpose temporary working space during directory sorts and file operations. The common area is managed internally within XPRESS, and the user never needs to worry about which section of the common area is being used at any given time for a specific function.

A portion of the common memory area, the text buffer, is used to receive all terminal transmissions from the remote computer. A series of pointers to text within the buffer allows the user to review terminal activity even after it has scrolled off the screen; automatically capture text for use with systems that do not support protocol file transfer; and provide buffered printing of the text, even if the printer runs slower than the communications line.



XPRESS allows these three functions to operate in the text buffer without error or interference with each other. The text buffer typically takes up as much memory as is available after the program is loaded. When the text buffer is being used only as a review buffer, then the buffer simply wraps around to the beginning as soon as it fills; the last buffer of information is always available for review.

When the text buffer is being used for nonprotocol file transfer, the Capture pointer keeps track of the contents of the buffer. As soon as the buffer is about 95 percent full, XPRESS stops the transmission of additional characters (using XOFF) and saves the buffer to a specified file before allowing the file transmission to restart.

When the text buffer is being used as a printer buffer, XPRESS sends characters to the printer while buffering incoming characters. If the printer is either busy or off line, XPRESS continues to fill the text buffer and resumes printing when the printer is again available. If the printer is so slow that the text buffer gets nearly full, XPRESS requests the remote system to suspend transmission by sending the appropriate character, usually XOFF (decimal 19).

Terminal Emulation

XPRESS emulates a DEC VT100 terminal in those modes possible on each specific microcomputer. These may include cursor position, cursor up, down, forward, and backward, device status report, cursor position report, save cursor position, restore cursor position, home erase to end of screen, erase to end of current line, set graphics rendition, set mode, reset mode, and keyboard key reassignment as specified in the ANSII terminal specification. The VT100 was chosen based on the large volume of software written for that terminal and its popularity in the mainframe marketplace.

File Transfer Protocols

The CRC protocol used in this implementation is an extension of that proposed by Ward Christensen in his Modem series programs. It is extended to allow the transmission of batches of files and to use the CCITT 16-bit CRC algorithm. The CRC protocol was chosen for its superior error recovery and also because a large number of microcomputer bulletin board systems already use it for file transfer.

The algorithm (x16 + x12 + x5 + 1) is specified as providing detection of all single- and double-bit errors, all errors with an odd number of error bits, all burst errors of length 16 or less, 99.9969 percent of all 17-bit error bursts, *and* 99.9984 percent of all possible longer error bursts. (See *Computer Networks*, by Andrew S. Tanenbaum, Prentice-Hall, 1981.) The CRC is set to zero at the start of each packet received and then accumulated for each data byte. At the end of the packet the CRC is updated with the MSB of the CRC and then the LSB of the CRC. The CRC should be zero at this time if no error occurred in transmission.

char 0	Start of Header (SOH-decimal 1)
char 1	Packet number modulo 64
char 2	Complement of packet number modulo 64
char 3-130	8-bit data bytes with no encoding of any kind
char 131	Most significant 8 bits of 16-bit CRC per CCITT-16
char 132	Least significant 8 bits of 16-bit CRC per CCITT-16

Table 1.

Data are transferred between systems by taking a sector of information from the disk and surrounding it with identifying information as well as information needed for error recovery. This block of data and administrative information is often called a packet. The actual packet format is as shown in Table 1 (above).

The ACK character (6 decimal) is used as a positive acknowledgement, and the NAK character (21 decimal) is used as the negative acknowledgement. The receiver initiates the connection by sending a capital "C" character (67 decimal) until it receives the first packet. This "C" character signals the transmitting machine to use the CRC protocol; it then uses the ACK and NAK as detailed above to confirm each packet. Packets refused by the receiver are retransmitted up to five times before giving up. Fatal errors are communicated by sending the CAN character (24 decimal) three times to the opposite system.

The file name packet is a special packet differing from a standard packet only in that it is null filled after the file name in the data area. A data packet filled with all nulls indicates the end of the batch of files being transmitted.

Note that the CRC protocol requires the use of all eight data bits and will send ACK, NAK, CONTROL characters, and characters with values in excess of 128 decimal during the normal transfer of ASCII and binary information. This protocol, therefore, cannot be used in systems that restrict the use of 8-bit characters in any way.

The XNET protocol is a fully packetized protocol useful in cases where the line will not allow all characters to pass freely. It is a proprietary protocol that maps characters into the dense graphics printable character set, thus allowing transmission over restricted lines. Packet size is restricted on configurations using the DECsystem-20 due to the inability of the DECsystem-20 to handle packets of data sent in bursts of greater than 94 characters. This protocol has receiver-defined parameters, including packet size and pre-packet and post-packet terminators, making it suitable on a large number of remote systems.

Related Software

Teachers today have a huge selection of software to choose from and often may lack the time to personally try out each package. We felt that a means to quickly select and review important facts about each available package was needed. A data base of software packages was developed, which provides the teacher with a means of obtaining a quick overview of all packages that meet specified selection criteria. The user may then request additional detailed information about any software package.

An example of how this software library application works might be helpful at this point. Let's suppose that a math teacher has an Apple computer and wishes to provide instruction to the students on how to find the area and volume of common shapes and solids. The teacher would use the Apple computer to communicate with the mainframe in terminal mode and would access the software library application program. Next, the teacher would request information on those software packages that run on an Apple, whose subject is math, grade level is sixth through ninth, and that are either available at no charge or commercially available.

The data base would be scanned, and information on the 15 programs in the data base that meet these criteria would be shown in summary format to the teacher. The teacher would then request specific information on three packages that seem to be most appropriate to the needs for the class. Additional information such as cost, system requirements, where the package could be obtained, and comments by other teachers on the package would then be shown to the teacher. The teacher could then elect either to download the desired public domain software directly using XPRESS or to contact the distributor for commercial packages.

Future Directions

It is apparent that school districts will continue to migrate toward micros

(Continued on page 31)

Communications Protocols:

Theory and Practice

by Leslie Brooks

Leslie Brooks, Computing Center, Florida State University, Tallahassee, FL 32306.

Copyright © 1983 by Leslie Brooks. All rights reserved. Permission is granted for non-commercial use and distribution. Commercial use without the author's written permission is forbidden.

How many of us have not had the frustration of trying to get some data from a friend, or some other micro or mainframe, only to discover that your machine's MICRO-COMM does not communicate very well with the mainframe's TSO or your friend's PC-LINK? By "communicate" I mean much more than just acting as a dumb terminal; I include full file transfer in both directions. The problem is an incompatibility of *protocols* between the machines and communications packages involved. In order to solve the problem, we need to decide just what a protocol is, how it works (or doesn't work), and what some typical ones look like.

Starting Simple: The XON/XOFF Protocol

Simply put, a protocol is an agreed-upon set of rules for two computers to communicate with each other, a "verbal handshake" between the computers, if you will. The simplest such protocol, and one with which most of us work every day, is the XON/XOFF (or DC1/DC3) protocol. Many terminals, such as the DEC VT-52 and Zenith Z-19, use this protocol to control the flow of data from the microcomputer. This illustrates one of the first goals of any protocol: To control the flow of data.

The XON/XOFF protocol uses the special characters XON (control-Q) and XOFF (control-S) to turn on and off the flow of data from the other end. CP/M uses control-S (XOFF) to stop the flow of data to the console, and any other key to restart it – a simple implementation of the XON/XOFF protocol. Printers often use this protocol to prevent their buffers from overflowing; a printer will receive data at 9600 baud until its buffer is about three-fourths full. At this point it will send an XOFF to the microcomputer, and the microcomputer will (hopefully) stop sending data. The printer will print for a while, until its buffer drops to only one-fourth full. The printer will then send out an XON, and the microcomputer will again send it data.

In our hypothetical case everything worked exactly as it should. In the real world things are not so simple and problems often develop. For example, suppose that your cat pounces on the printer cable just as the printer sends out the XOFF. You live in Nevada and the air is very dry, so the cat has built up a static charge as it crept up on the cable. This static electricity discharges into the cable and the XOFF is drowned in noise. Your micro never sees the XOFF and continues shoving out data at 9600 baud. Hopefully the person who designed your printer's software anticipated this, and your printer sends out another XOFF when its buffer has only 100 bytes left. This illustrates the second goal of any good protocol: Error tolerance. Herein lies a *very* deep swamp!

Admittedly, most printer designers probably considered the case we looked at above, but how about a very similar case? Again, picture your cable getting a noise burst, but just as the XON goes out. Your microcomputer never sees the XON and doesn't transmit any more data. Most printers will allow this situation to continue until their ribbons rot and fall off! A clever designer might have the printer send out XON's every ten seconds or so if the buffer is empty and it is not receiving data. This is a third goal of a good protocol: To anticipate and prevent deadlock conditions. Not every protocol succeeds here, and the success or failure may often depend as much on the implementor as on the protocol.

Micro to Mainframe and Back Again

If you have stayed with me this far you can already understand most of the reasons why your microcomputer cannot talk to your friend's: they don't use the same protocol. This is also the problem with trying to communicate with many mainframes. In the real world, then, how do we surmount the problem? Well, there *are* some pretty standard protocols in wide use. For example, the CLINK protocol developed by Larry Hughes and first used in his CLINK communications package is also used in Crosstalk and in Larry's later package, MITE. (See the SEND/RECEIVE article in the August 1982 issue of *Dr. Dobb's* for some of Larry's earlier work.) The XMODEM protocol, developed by Ward Christensen, is in widespread use on many different machines and in many different implementations, and is in the public domain. Let's take a quick look at it.

XMODEM

The XMODEM protocol is a reasonably good protocol, widely used, and particularly suited to use on microcomputers. A typical transmission would look like

| | | |

Of Transmission) to indicate that no more data remains after the last block has been acknowledged.

XMODEM, then, meets most of the requirements of a good protocol; why doesn't everyone use it? Well, some companies like to have their own communications packages, so they produce a proprietary protocol. Mainframes do not use it because they use other protocols, most of them much older than XMODEM. However, almost all CP/M bulletin board systems use it, and some packages (such as MITE) support it in addition to their own protocols. Even better (if you need to talk to minicomputers), there is a package called XCHANGE11 that implements the XMODEM protocol for DEC PDP-11 and VAX minicomputers. So XMODEM solves many problems; where does it fall short?

Mainframes

If you want to send files to a mainframe, or a minicomputer other than a DEC, then XMODEM will not help you. These machines just do not speak the correct protocol. Our solutions now are to change the protocol to one the mainframe understands, or implement one we know on the mainframe. Larry Hughes has a Fortran program which can be installed on a micro, mainframe, or mini to talk to his MITE package. He will send you a copy free if you ask for it when ordering MITE. This is a pretty good deal since it lets you talk to many machines for which no other simple solution is available. However, this gives you fairly slow communications – not very good for moving large amounts of data. Where are we to find a really good solution for micro-to-mainframe communications?

I mentioned earlier that many mainframes use protocols that were developed long before XMODEM was around. Some of these protocols show their age; XMODEM is better by far. Others were pretty well designed and offer capabilities and performance well beyond anything XMODEM is capable of. The mainframe protocols normally use a cyclic redundancy check (CRC) rather than a checksum; this gives much greater confidence in the accuracy of the received data.

Most of the common protocols allow blocks of data to be any number of bytes (or even any number of bits) long. This is more efficient than fixed block sizes, but introduces other problems. For example, because the block can be of any length, the protocol must either transmit the length in advance or mark the end of the block in some way. If the end of the

How Accurate Is Accurate?

by Leslie Brooks and John Rasp

When you are typing data or commands for the mainframe to process, you can tell immediately if you get a transmission error because the wrong character shows up on the screen. You then backspace over it and retype it. When large amounts of data are being sent between machines, this method of error checking is inefficient because of its character-by-character nature. Thus the protocols that have evolved usually provide some other means of detecting errors, such as checksums or cyclic redundancy checks. The next question is, "How good are these methods of error detection?" The answer can become very complicated; people have earned Ph.D.s in math for research in this area. I put this problem to a friend of mine who is doing doctoral work in statistics and he pronounced it "interesting." After a few days of study and research he told me that if we make a few assumptions, we can get a reasonably simple answer.

For the 8-bit checksum, there are actually three common ways of calculating it; they do not produce identical results. The first method is simply to add all of the data bytes together, modulo 256, and take the result as the checksum. The second method is the same, except that any carries generated are added back into the sum. The third method is to exclusive-or all of the data bytes together (sometimes called a longitudinal redundancy check). The best accuracy one can, in theory, hope for with an 8-bit checksum is to miss one error in 256. In practice, however, things may be less accurate, depending on the method used and the context in which the error occurs. Any simple calculation of the accuracy of these error-checking methods assumes that:

(1) There are about the same number of ones as zeroes in the data. The mathematics becomes much worse but the probability of detecting the error improves if this assumption is not true.

(2) The changes are independent of each other. This is not true – because phone line noise occurs in bursts, the fact that bit n was clobbered greatly increases the probability that bit n+1 will be clobbered also. My friend feels that this will not affect the result, particularly if assumption (1) is true. If assumption (1) is false *and* the changes are dependent, the math really gets hairy.

(3) There is an equal probability of changes in each direction. That is, saying that bit n was clobbered means the receiving system will see a one bit exactly half the time. This assumption should be true of noise, but may not be true of hardware faults. If it is false, the probability of detecting the error increases, but it is very difficult to say by how much.

After all of those caveats, if we assume that, on the average, ten bits get hit in any particular bad block, then the probability of missing an error is between one in 142 and one in 147. The longitudinal redundancy check misses approximately one error in 142. My friend was able to calculate this mathematically, but the math involved with the two addition methods was so complex that a Monte Carlo simulation was set up to estimate the accuracies. If the 8-bit checksum is performed by the simple addition of data bytes, modulo 256, approximately one error in 147 will be missed. Our confidence factor here was roughly 95 percent. When the carry is added back into the sum, the checksum is slightly less accurate.[1] This is acceptable for noncritical applications.

The cyclic redundancy check is a better error checking mechanism than the checksum. The mathematics are quite interesting; they are based on treating the message as a huge integer. The message is then (for a standard CRC-16) multiplied by 2^{16} and then divided by 11000000000000101 (the generator polynomial, 17 bits long). The 16-bit remainder is then attached to the end of a message as the error check code. This process is repeated on the receiving end to check for errors.[2]

The standard CRC-16 (sixteen-bit CRC) will detect all errors of 16 bits or less, and 99.955 percent of all errors of more than 16 bits.[3] This means that in the worst case (more than 16 bits in error) we will fail to detect the error only one time out of 2200. If we again assume that on the average only ten bits are changed, the probability of not detecting the error becomes even smaller.

References

[1] John Rasp, Statistics Department, Florida State University, Tallahassee, Florida 32306.

[2] John E. McNamara, *Technical Aspects of Data Communications*, Digital Press, pp. 145-158.

[3] McNamara.

block is marked, then this special end-of-block character must not appear in the data. This problem is usually solved by defining a transparent mode, in which the data is altered to remove any special characters before it is transmitted. For example, in HASP a DLE (Data Link Escape) character in the data would become DLE DLE, and the receiving station would throw away the first DLE. The common mainframe protocols are not simple!

These protocols usually require synchronous modems, and 2000 baud is the slowest speed commonly available – 4800 baud is very common, and 9600 baud is not unusual. If you really need to move a lot of data, this is the way to go. This approach is not cost-competitive with XMODEM or MITE, but it offers tremendous performance and easy entry to mainframe environments where the other packages are not available.

2780/3780

A typical mainframe protocol would be 2780/3780, an IBM protocol developed many years ago and now in widespread use by many other companies. It is available on virtually every mini and mainframe in existence. At 2000 baud, XMODEM's 128-byte blocks would take only half a second to transmit, so 2780/3780 allows for larger blocks. The larger blocks give much better line utilization, particularly at higher baud rates. However, 2780/3780 is not a simple protocol; it requires larger buffers than XMODEM (and uses much more memory overall). There are also plenty of places where the implementor can make mistakes. Still, it is a good protocol and very useful for moving large volumes of data around. It is primarily designed for remote batch stations and for moving data, not for interaction. The major disadvantage of this protocol is that only one thing can be happening at any given time. For example, the user can be printing a file from the mainframe, but cannot be downloading a file to disk at the same time.

HASP

HASP is another IBM protocol; it is very nearly a superset of 2780/3780. It is widely available on mainframes and minis of all types. HASP provides capabilities that are really incredible (the protocol defines seven printers, all of which can be going at once!). HASP also compresses data before it is sent, increasing the effective transmission speed. A good implementation of HASP will eat 48K of RAM in an instant, and one could easily use 64K or even more (if it allows large buffers, is fully menu driven, has on-line help, etc.). Most implementations on micros leave off a few things – like five or six of the seven printers. (When was the last time you had the need to run three printers simultaneously from your micro?)

HASP is also very complex, and there are many places to make mistakes. Again, a top quality implementation will anticipate and correct for mistakes in the *other* machine. I helped develop a package called HASTE for the computing center where I work and I know that the other implementations do make mistakes sometimes. In some cases (not all) clever programming on the micro end will correct faults on the mainframe end. For example, HASP compresses data before it sends it; in theory a single 400-character block (a common size) could expand into 16K of actual data! Needless to say, if you try to expand the received data in memory, you can get yourself into real trouble. HASTE expands the data only when it reaches its ultimate destination – the console, the printer, or disk. We have seen a mainframe implementation, however, which tries to do it in memory – we know because we blew it out of the water with a large compressed block!

HASP also defines control bits to control the different data streams (one stream for each of the seven printers, plus console, card reader, and card punch). These control bits may be transmitted with any block of data or as a separate message if there is no data to send. Thus when the micro's printer buffer fills up, we send out the FCS bits saying, "You can't send me any more printer data." Later, when the buffer empties, we send another message saying, "Printer stream one is now available again – send more data." If the micro receives a NAK (negative acknowledge) it must retransmit its last block.

What is not clear from the documentation is whether the retransmitted block should contain the previous FCS bits or the current FCS bits. If we send the previous FCS bits (saying "Send more data"), but our buffer is now full and the mainframe believes that FCS bits are always correct, then it will send data that we have no room for. If we send the current FCS bits, and the mainframe believes that FCS bits stay with the block (i.e., may not be current), then it will not act on FCS bits sent with a retransmitted block (they may be out of date). This means that we can send FCS bits saying, "Send me more data," and the mainframe will ignore them. We will then wait forever for the mainframe to send more data. The solution to this is to treat the FCS bits as always current, and retransmit them until they eventually go out on a block that is not itself being retransmitted. This will work with either implementation on the mainframe. Not every implementation (micro or mainframe) concerns itself with little details like this!

To give you a feel for the speed of a protocol (and package) like this, I recently downloaded 4.5 megabytes of data from our mainframe to the hard disk on a Kaypro 10. Using a 2000 baud modem it took just five hours.

HASP is again designed more for moving data than for interaction, but it allows concurrent operations. A single user can in theory be printing a file from the mainframe, sending the mainframe a file from disk, receiving some data and storing it on disk, and typing in some commands for the mainframe, all at the same time! Not all implementations will support this in the same way, but most provide at least some of these capabilities. HASP makes for a wonderful (although expensive) micro-to-micro file transfer environment. Because of the multi-stream capability you can type messages to your friend on the other end of the line at the same time that you are sending him or her the latest copy of the new utility you are working on. No more need for a second phone line or hanging up and redialing.

3270

As a last example of a popular protocol, 3270 is yet another IBM protocol that sees wide use. It is, however, not widely used outside of IBM sites. This protocol is designed for interactive work at a terminal; it thinks in screens. It is synchronous like HASP and 2780/3780, and can be either bit or byte oriented. HASP and 2780/3780 deal strictly with bytes – 3270 can transmit three bits as easily as eight. This protocol is not used much for moving data between systems, primarily because that is not what it does best. The 3270 protocol contains information on cursor positioning, screen clear, and other terminal-related things.

Conclusions

Communications is a very large, very deep swamp, and it contains many alligators. If you jump in with no preparation, you will get eaten alive. However, with a bit of thought toward what you wish to achieve, you can get through it successfully.

If cost is of utmost importance, look at XMODEM. If you have any sort of common micro, XMODEM is probably already available for it for free.

If you need to talk to a single mini or mainframe, and need to move moderate (less than 500K characters per week) amounts of data, take a good look at XCHANGE11, MITE and its Fortran friend, or some of the other similar packages which are available.

If you need to move large amounts of data, or need to talk to several different mainframes or minis, investigate the HASP and 2780/3780 synchronous packages. These are more expensive initially, but pay for themselves with their tremendous

(Continued on page 80)

Unix-to-Unix Network Utilities

This article examines a relatively unknown yet powerful set of asynchronous Unix-to-Unix network utilities initially developed for internal use at Bell Labs and now available on most of today's Unix micro machines.

These network utilities provide perhaps the most standard (if not the simplest) method to move files and programs from one Unix machine to another. More importantly, a manageable and inexpensive network can be created to perform a variety of chores such as remote spooling, central filing and backup, remote data collection, electronic mail, and so on.

There are two basic utilities: uucp and uux. Uucp (Unix-to-Unix copy) is intended to work much like the Unix file copy utility, cp. The only difference is that the source and destination files also include the name of a Unix system on the network. The uucp syntax is:

uucp [-options] sys1!src sys2!dst

such that **src** and **dst** are source and destination filenames, and **sys1** and **sys2** are the designated machines on which the file exists or to which the file is copied. The entity in the square brackets is optional.

For example, the command

uucp A!file1B!file2

will copy **file1** on machine **A** to **file2** on machine **B**.

Uux (Unix-to-Unix execution) allows commands to be initiated on the local machine and executed remotely on another machine. A request to execute a command is sent to the remote machine, and if the request can be honored (i.e., the program exists on the remote Unix machine and is executable), the remote machine will try to execute the command.

The uux syntax is:

uucp A!file 1 B!file2

cmd2 | uux-sys1!cmd1 [sys2!files]

or

uux-sys1!cmd1[sys2!files] < [sys3!]file

In the first case, uux will execute the command **cmd** on the system **sys1** and, if necessary, will first copy the required **files** from system **sys2** to system **sys1** to be available to **cmd** on its command line. The next two cases are much the same except that uux's standard input is redirected to come from a Unix pipe (the second case) or a file (the third case); the rest of the arguments function the same as in the first case. Note again that the entities in the square brackets are optional.

The important thing to note about the last two cases is that uux's standard input is copied to the remote machine **sys2** along with the request to execute **cmd1**. Once on the remote machine, **cmd1** and its command line are reconstructed for proper execution.

The Unix-to-Unix network utilities include a number of other utilities that perform, among other things, the actual machine-to-machine communications (protocols, handshakes, log-in procedures) and the actual remote machine execution of the requested commands. Uucico (Unix-to-Unix copy-in copy-out) and uuxqt (Unix-to-Unix execute), respectively, perform these chores—typically in the background.

Each time a uucp or uux command is issued, a group of special "work" files is queued in a spool directory, and uucico is run in the background to process these files. The "work" files contain all kinds of information about which files are to be copied and to where, how the files should be copied, and how to execute the requested command. Since uucico is run in the background, the user does not need to wait until the actual work is completed before receiving control of the Unix shell again. As uucico works, it posts status checkpoints in a log file.

After finding a work request and checking permission for the request, uucico attempts to log on the remote machine. The entire log-in procedure is user-defined when the uucp system is first configured and can be tailored to perform almost any dial-out/log-in sequence.

If the Unix log-in prompt is successfully reached, uucico logs on the remote machine as a user named "uucp." Logging in as "uucp" causes the remote machine to execute a uucp command, which in turn forks a uucico command in "slave" mode; the uucico process that made the call to the remote machine runs as a "master." The slave process initiates the handshake and the master acknowledges. They agree upon a protocol and the master begins sending requests.

After the master has exhausted its requests, the uucico roles are switched—the master becomes the slave and the slave becomes the master. If the called machine has work bound for the calling machine, the new master sends its requests to the new slave. This continues until neither machine has any more work, at which point they hang up.

Since Lehman Brothers has several in-house Unix machines, we felt that with uucp and uux we could take these off-the-shelf utilities and build a network. Our first application was to create a printing network (using primarily uux) in which any user on any machine could access any printer on any machine.

Building a network was not as easy as it sounds because: (1) we wanted the existence of the uux utility to be completely transparent; (2) we wanted the print command interface to be consistent; and most difficult of all (3) we wanted the uux command to work unattended even in the face of such things as log-in failures, locked devices, and a few minor but potentially crippling uucico bugs that are easily handled manually.

The first and second goals were met by constructing a virtual printer. This is a channel through which the print data is piped to a uux process; uux then routes the data to the target machine.

Achieving the third goal took much more effort. Since uucico runs in batch fashion, the user cannot interactively assist uucico during the processing of the work files and log-in procedures. Some problems can be anticipated, but if the log-in fails, uucico simply records the failure in the log file and dies; the user receives no message and may never know if the request was fulfilled. Moreover, once the log-in fails, uucico locks all calls to the remote machine for about an hour. Another attempt to fulfill the request is not possible until uucico is restarted manually or until uucp or uux is issued again—after the hour of lockout time.

Worst yet, uucico sometimes dies for no apparent reason, even though the log-in was successful! When this happens, uucico fails to reset the permissions on the /dev/tty files and the spool directory (/usr/spool/uucp).

To resolve these problems, we first carefully chose a log-in sequence that has about a 7 to 10 percent failure rate. Next, we wrote a daemon that wakes up whenever files are being spooled remotely. This small program can reset the permissions on the /dev/tty files and spool directory

by Ron Coleman

Ron Coleman, Advanced Systems Group, Lehman Brothers, Kuhn, Loeb, Inc., 2 Broadway, New York, NY 10004, (212) 839-0965.

and knows how and when to remove the lock files and automatically restart uucico.

Figure 1 (below) illustrates this network. The user sees only Lp1, Lp2, Lp3, and Lp4. If a user sends a request to a local printer on his machine, the request is honored directly. Whenever a printer is specified that is not on the local machine, a special preprocessor shell script adds one line of routing data to the beginning of the input file, and the request is given to the virtual printer. The virtual printer reads the first line of the modified input stream and integrates that line in a uux command string. The actual print data is then piped to the uux process that uses the routing string to direct the print request to the remote machine. Although a remote print request has yet to fail, we estimate the average failure rate to be about 3 to 5 percent. DDJ

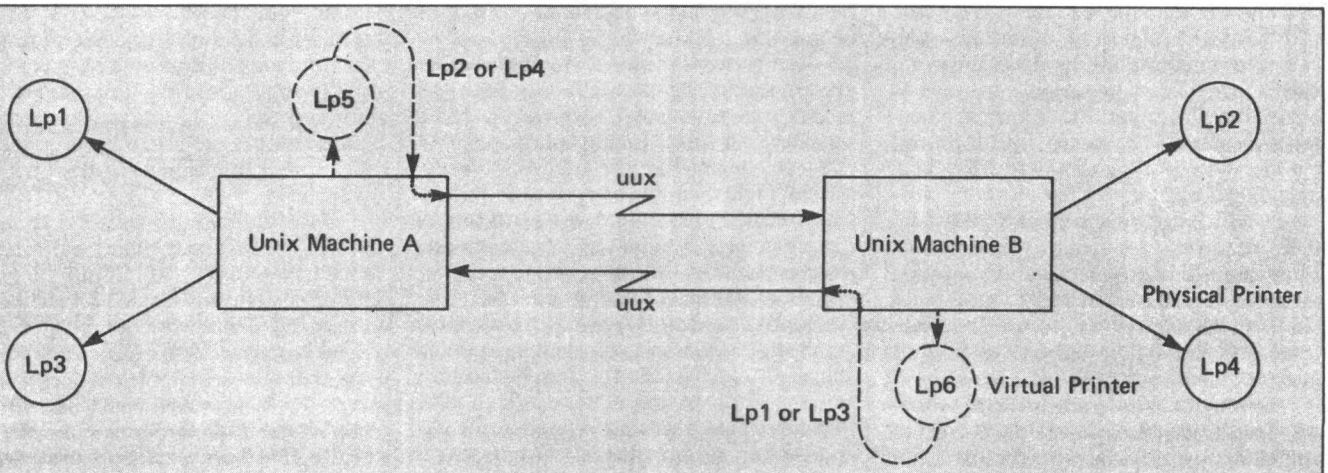


Figure 1.
Unix-to-Unix Network:
Physical/Virtual Printer Relationship

(Continued from page 23)

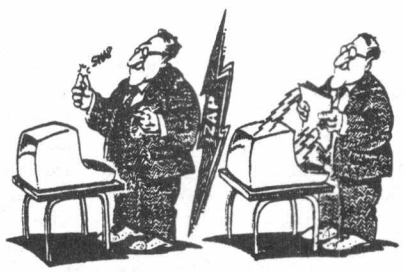
for many applications and accordingly will make less use of the mainframe. Putting into practice one of our major goals, which is to put data processing control into the hands of the users, we created the PIPELINE system. PIPELINE is a user-oriented system that works with a multiplicity of microcomputers.

Using this system, users decide what data they need and how they want it parameterized for their special reporting or analysis needs. Then, using their micro as a dumb terminal, they access the mainframe and ask for PIPELINE. They are presented with a menu that asks them what kind of data they want. If, for example, they want personnel data, they are routed to a data dictionary that lists all of the personnel data elements on the system, each sequentially numbered with a one-line descriptor. Then they choose by number (such as 33, 96, and 101) which data elements are to be extracted.

If additional data from the financial system are needed to integrate with the personnel data, then the users ask for the financial menu and go through the same process to extract the needed financial data. When the users have extracted the defined data, they indicate the data should be downloaded to their micro. While interrogating the mainframe system for information, users may use Boolean Logic selectors such as greater than (>), less than (<), or equal to (=) in specifying what kinds and ranges of data they want (i.e., select teachers whose subject = math and who have > 5 years of experience).

Once this data has been downline loaded in a format that the users' micro can handle (using XPRESS as the vehicle), it then may be further manipulated using a local data base manager.

A complementary approach is the use of a software package that allows a large mainframe to run microcomputer software, including the popular CP/M operating system and the large quantity of inexpensive programs that run under it. With no additional hardware investment (such as the purchase of a micro), users may use their current terminals and operate them as if they were a fully configured microcomputer; at the same time they have the full power of the mainframe available.



The users could, of course, also use a micro to downline load any information to their floppy disks and run whatever programs they wished independently. The best of both worlds, this approach allows CP/M and CP/M-based programs, as well as stand-alone programs, to run on the DECsystems. Except for their speed and the fact that they are easily accessed remotely over telephone lines and networks, such programs run exactly as they do on a microcomputer.

Investigation is being done on techniques for data compression and encryption to meet the needs of high-volume data transfer with security. Among the techniques being explored are Hoffman coding, repetitive character encoding, use of private keys, and more efficient protocols.

The future is bright with promise for the connection of today's new series of microcomputer to the traditional commercial mainframes to provide the strength of the mainframe with the flexibility and local control of the microcomputer.

DDJ

Virtual Personal Computer

The Virtual Personal Computer (VPC) project is an effort to develop a general architecture and specification for a personal computer that can be integrated with networks and larger computer systems. This article concentrates on the overall philosophies and architecture of the VPC.

The primary goal of the VPC project is to integrate personal computing and time-sharing. Time-sharing systems and personal computer systems have been based on fundamentally different philosophies. The Unix time-sharing system was designed to support a large number of dumb, full-duplex terminals. Typical Unix users are accustomed to the normal delays and slow communication links associated with such systems. Personal computers, on the other hand, have been designed as dedicated nontime-shared systems offering users unbelievable real-time response. Personal computer users that have never used a time-sharing system are being spoiled by this dedicated processing capability.

Software developed for time-sharing systems may be incompatible with personal computers and vice versa. As an example, in the Unix time-sharing system the concept of blocked I/O is used extensively. When a process wants to read information from a user's terminal, it issues a **read** system call. If the user has not typed any information, the process becomes **blocked**. The process will wait indefinitely until input is generated. While the process is blocked waiting for terminal input, it is not allowed to do any computation or read any other I/O devices. This blocking frees the main processor, which allows other users to share the same processor (thus, the term time-sharing).

Personal computer software is often designed without the concept of blocking. Instead, if input is desired from a user terminal, a check is done first to see if input is available. If input is available, a read is requested. The read system call returns immediately with the information without blocking because the check had indicated that data was available. If no data is available, another check can be made, and the user program will loop indefinitely waiting for inputs. If other I/O sources need to be serviced, the user program can check these multiple sources and read any that have data waiting. Keyboard input can be checked, a printer can be controlled, and a communication link can be serviced with precision under control of a user program.

As time-sharing and personal computer environments begin to converge, we must be aware that an indiscriminate merging of techniques from each environment may not yield an optimum situation.

The goal of the VPC project is to integrate the best of personal computing with the best of time-sharing. The VPC handles all local real-time-sensitive activities to give the user instant feedback. Programs running on the VPC are designed like classic personal computer programs with the check/read looping methods. Because it is assumed that a dedicated processor is available in the VPC, no concern is given to the processor cycles wasted using this methodology.

The application programs resident on the host system, however, are designed using the normal time-sharing blocked I/O techniques. Messages are exchanged between the VPC and the host at a rate compatible with these methods.

by Jim Fleming

Jim Fleming, Unir Corporation, 5987 E 71 St., No. 106, Indianapolis, Indiana 46220.

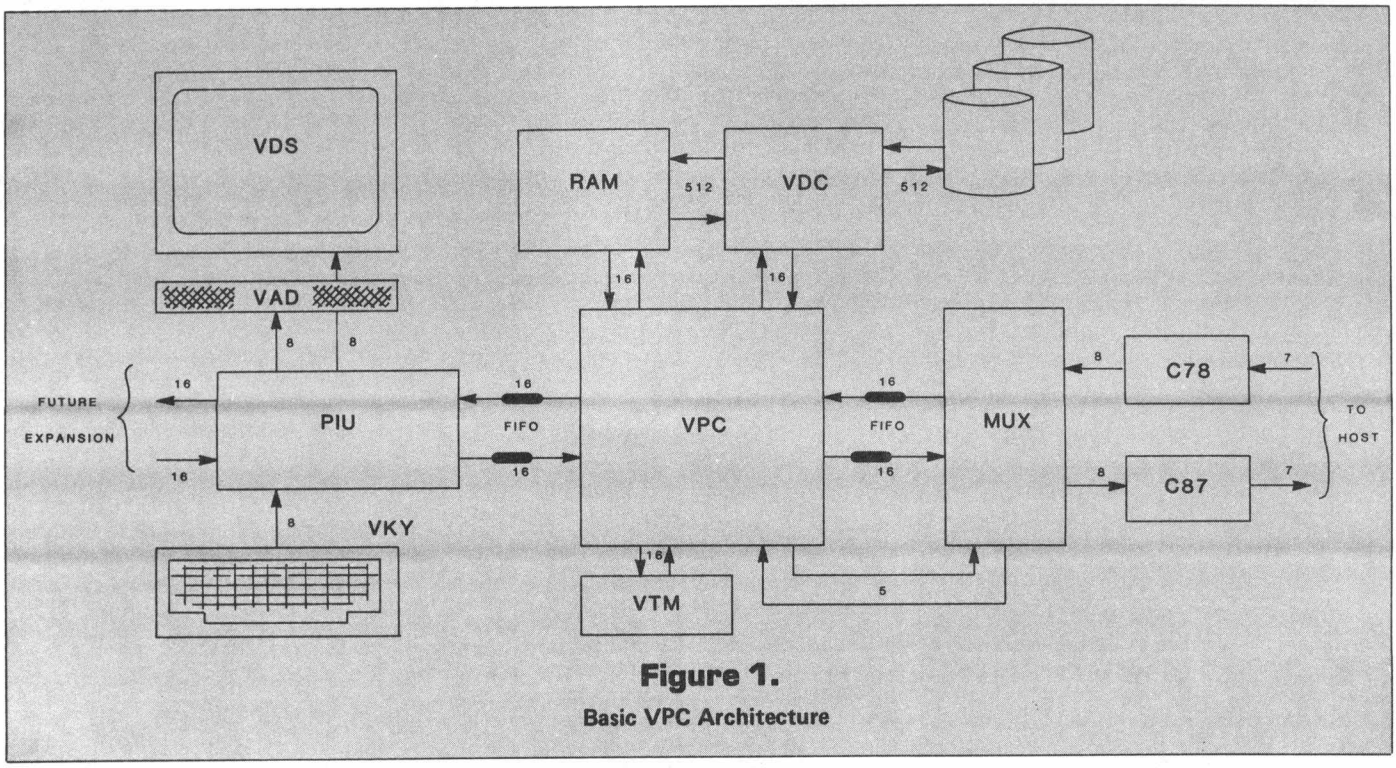


Figure 1.
Basic VPC Architecture

In systems like Unix, no change needs to be made to the application programs. The functions normally performed by the device drivers are "remoted" to the VPC. A simple change to the existing Unix device driver (which reduces its size) makes the VPC transparent to the application. The user is also unaware that functionality has been moved from the host into the VPC. High-speed real-time response is now possible so that personal computer users are not penalized by integrating their system with a time-sharing system.

Basic VPC Architecture

The basic VPC architecture is shown in Figure 1 (page 32). A process(or) is connected to the host system via a bi-directional 16-bit data path and a 5-bit control path. The 16-bit data path is called the remote I/O channel, and the 5-bit control path is called the remote control channel. The process(or) is interfaced to the user by another 16-bit bi-directional data path called the local I/O channel.

The process(or) is a 16-bit machine that executes one program residing in a 64K address space. The process(or) can be controlled by the host via the remote control channel. The process(or) can send and receive data to the host or user over the respective I/O channels. Local disk storage is supported using a hierarchical file-naming structure. An interval timer and time of day clock are also supported.

Character-oriented peripherals are connected to the local I/O channel. The upper eight bits of the local I/O channel are decoded as a device address, and the lower eight bits carry data to and from the device. Many protocols can be supported by these character-oriented peripherals.

The architecture of the VPC is based on the layered approach used in many data communication systems. Layering allows various parts of a system to be specified independently of the other parts. Changes can be made in a layered system with the confidence that the entire architecture will not fall apart.

Four distinct layers are evident in the VPC architecture. The four layers can be seen in Figure 1 by imagining that vertical lines separate the major boxes. Working from right to left, one can see the link layer, the multiplexer layer, the process(or) layer, and the user layer.

The Link Layer —
7-Bit, Asynchronous, Even Parity Version

The primary purpose of the link layer of a VPC is to guarantee that 8-bit bytes can be transferred in both directions in an error free (or error unlikely) manner. A variety of link layer configurations can be developed. Only the asynchronous parity-checked configuration will be discussed here because this is commonly used on personal computer systems.

In Figure 1, the link layer consists of the boxes C78 and C87 plus the hardware necessary to transmit and receive data to and from the host. In a 7-bit, even parity, asynchronous transmission system, a UART and modem are usually used as the primary hardware components. In this type of system, only seven data bits can be passed across the link. The software modules (C78 and C87) are used to convert one or more 7-bit data items into an 8-bit byte. The 8-bit bytes are used to interface the link layer with the multiplexer layer.

Many schemes have been devised to convert 8-bit bytes to other forms for transmission through 7-bit channels. The most common technique is to split each 8-bit byte into two 4-bit nibbles and send each nibble as an ASCII character from the set (0-9, A-F). This method works fine but doubles the amount of transmission. A 4000-byte file requires 8000 bytes of transmission.

Other methods block multiple 8-bit quantities together and enclose the packet between a header and a trailer. The data (the 8-bit quantities) are wrapped inside the packet and flow through the channel in a semi-transparent manner. Blocking does not help the problem of pushing 8-bit packets through 7-bit channels. Blocking can also be inefficient when many small data messages are sent because of the header and trailer overhead.

The filters shown in Figure 1 have been designed to push 8-bit bytes through 7-bit channels without the need for blocking and without interfering with control codes commonly used for flow control (e.g., XON, XOFF). The C87 filter (see Listing One, page 46) can be used to convert any 8-bit stream into a 7-bit stream. The C78 filter (see Listing Two, page 50) will convert the 7-bit stream back to 8-bit format. Each of the filters reads the standard input and writes on the standard output and can therefore be integrated with other Unix tools.

The filters operate by starting with the assumption that only 96 codes (32-127) are available for data. The lower 32 are assumed to be control codes, and the upper 128 are lost to the parity bit (normally even). The 96 codes are split into two parts. The codes 32-39 are used to encode one of eight base addresses. The eight base addresses are all offset by eight to account for the common occurrence of ASCII data in the channel. The offset of eight is assumed by both filters and not sent through the channel. The codes 40-127 are used to encode an offset from the base address.

The base address is changed only when a code cannot be reached by the offset. If the base address has to be changed, one 7-bit data item is sent to change the base address, and one 7-bit data item is sent with the proper offset. If the base address does not have to be changed, then one 7-bit data item is sent with the offset. As long as the base address does not have to be changed, each 8-bit byte is encoded as one 7-bit data item. In normal practice one finds that ASCII text grows by about 25-30 percent and object modules grow by about 30-35 percent.

These filters are used to form the link layer processing in the VPC. The C87 filter processes incoming data, and the C78 filter processes the outgoing data. The normal Unix device drivers handle all flow control and do not interfere with data because the control codes are avoided. The 8-bit data emerging from the C78 filter are sent to the multiplexer and further decoded. When the multiplexer is ready to send an 8-bit value to the host, the C87 filter processes the byte. *All* data values (0-255) can be sent through this layer using these filters.

It should be noted that this link layer method is able to detect some errors in the transmission channel but is not able to correct any errors. This method can be used when a VPC is connected to a host via a hard-wired connection or a low-speed modem.

Future versions of the VPC will feature different link layer arrangements. Link layer methods for blocked asynchronous and synchronous transmission are being developed. These methods allow for error detection and correction via retransmission. They guarantee an almost 100 percent error free transmission. This change to the link layer will not impact other parts of the system.

Another item of note concerning the link layer is that, when a VPC is run resident on a single Unix system, no link layer processing is needed. The VPC multiplexer layer is connected via Unix pipes to the host multiplexer layer. Unix pipes are able to pass 8-bit bytes in an error free manner from one process to another.

As can be seen, a lot of options are possible in the link layer portion of the VPC architecture. Because of the layered design approach, changes in the link layer do not affect the other layers. In fact, in most sophisticated networked systems the link layer can be replaced by existing low-level transmission methods. If a system already supports X.25, ETHERNET, or another proprietary protocol, the link layer function of the VPC can be performed by the resident transport mechanism. The only function that the link layer performs is to provide an 8-bit error free path with flow control. If this can be provided in another manner, one is free to use that method. The other layers of the VPC are not affected.

The Multiplexer Layer

The multiplexer layer is the heart of

the VPC-to-host interface. The sole purpose of the multiplexer layer is to encode the 16-bit data from the remote FIFO and the 5-bit data from the remote FIFO (discussed at greater length in the next section) and the 5-bit data from the remote control channel into 8-bit bytes. These 8-bit bytes make up the interface to the link layer. The multiplexer encoding scheme is extremely simple and is shown in Figure 2 (below).

When a 0XXXXXXX (0–127) code appears on the 8-bit channel, the lower seven bits are added to a previously loaded 11-bit register, producing a 16-bit result (see Listings Three and Four, pages 38 and 39 respectively, and the discussion on Levels of VPC, page 44). The 11-bit register is padded with five zeroes (on the right) before the addition is performed. The 2-bit overlap between the two values means that the 7-bit code is actually a positive displacement from a base that is always a multiple of 32. The 7-bit code can actually reach 128 values from a given 11-bit base. The 11-bit base is changed only if this 128 position window is exceeded.

Initially 00000000000, the 11-bit base can be changed using 10XXXXXX or 110XXXXX. The middle 6 bits of the 11-bit base register are loaded when a 10XXXXXX (128–191) is received. The upper five bits of the base register are loaded when a 110XXXXX (192–223) is received. No data are produced when either type of code is received; the 11-bit base register is changed and saved for future use. All multiplexers should initialize the 11-bit base register to ensure that a known state is set when the link is established. The default value at start-up is 00000000000.

The codes 111XXXXX (224–255) are used for a 5-bit control channel. The 32 codes that form the control channel are divided into 16 opcodes (1110XXXX 224–239) and 16 data values (1111XXXX 240–255).

The end result of the multiplexer function is to create the illusion that a 16-bit bidirectional data path and 5-bit bidirectional control path connect the VPC and the host. If one desires to build this arrangement using ribbon cables and large connectors, the multiplexer can be removed. The other parts of the system are not impacted except that things may run a little faster.

The 16-bit data path is used for all data traffic between the VPC and the host. No subchannels are assumed, and any 16-bit value can be sent through the channel. Subchannels can be created by conventions established for various applications.

A typical arrangement is to use the upper eight bits of the channel as a subchannel number and use the lower eight bits for data. This arrangement yields 256 independent 8-bit channels from the host to the VPC. The VPC may use a dozen of these channels for video windows, a couple for printers, and a couple for keyboards, and save the other 200 for future use. In noninteractive applications, 256 pipes can be established between the VPC and the host. This should be sufficient for most applications.

Another arrangement might be to use some of the high-order bits (five is a nice number) as a subchannel number and the other (three if five is used) high-order bits as subchannel data type indicators. For example, subchannel types for control and data could be determined by the type bits. With this arrangement, data arriving at the host can be classified as data or control and routed to the proper process or driver. Priorities can also be encoded in the upper bits so that channels can be handled with the proper urgency in the host. In all of these arrangements, eight free data bits are available in the low bits for total compatibility with existing systems.

The important thing to remember about the multiplexer is that its only job is to multiplex the 21 bits (16 data, 5 control) of information needed by the VPC process(or) into 8-bit bytes. No special assumptions are made about the 16 bits of data that flow through the multiplexer. Since the host has the ability to reconfigure the VPC process(or) via the 5-bit control channel, application-specific conventions can be established without impacting the multiplexer.

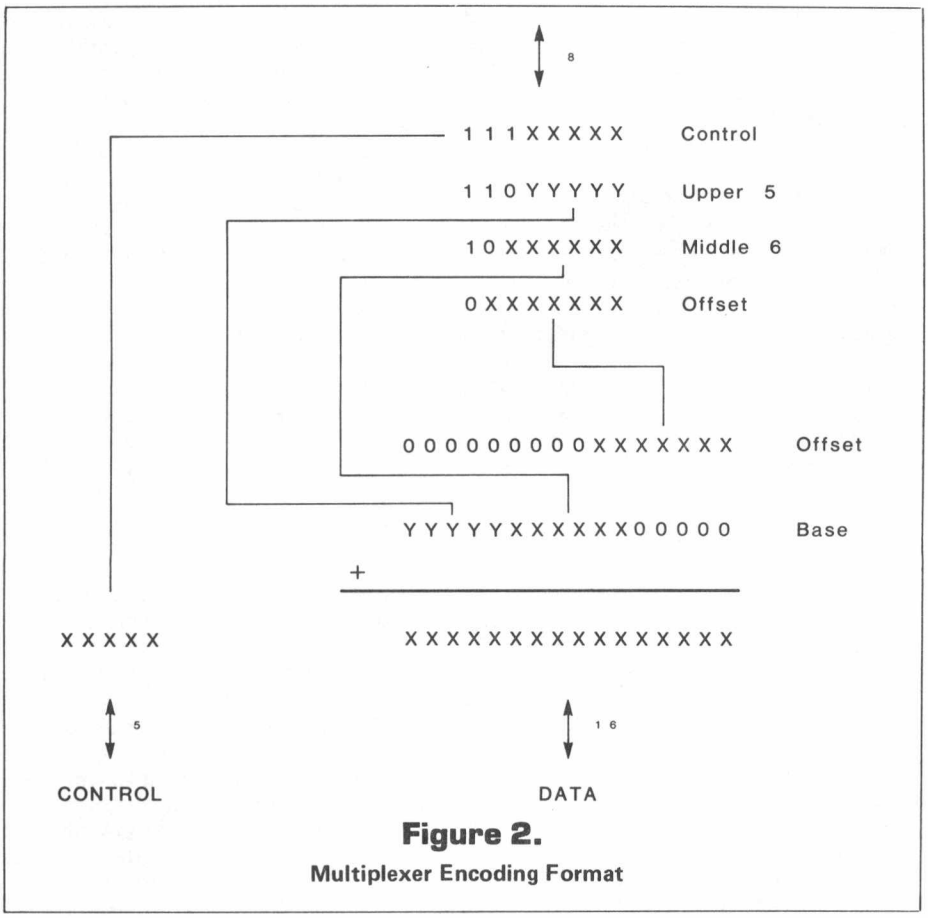


Figure 2.

Multiplexer Encoding Format

The VPC Process(or) Layer

The VPC process(or) layer interfaces to the multiplexer layer and the user layer via buffers called FIFOs (First In First Out). As the name implies, these 16-bit wide buffers are loaded and unloaded in the same order.

The remote FIFO interfaces the VPC process(or) to the multiplexer layer. Data written into the remote FIFO are eventually processed by the multiplexer and sent to the host. Data received from the host on the 16-bit multiplexer data channel are loaded into the remote FIFO and become available to the VPC process(or). The remote FIFO is bidirectional, and the data for each direction are kept independent of the other direction.

A similar arrangement exists for the local FIFO, which interfaces the VPC process(or) to the user layer. The user layer contains various device driver software modules and peripherals that interface with the user. The VPC process(or) can send and receive information to and from the user via the local FIFO.

The VPC process(or) is a classic 16-

bit processor with 64K of data space. Program execution normally begins at location 0. The VPC process(or) contains a program counter, stack pointer, stack frame pointer, argument pointer, and literal pool pointer.

The VPC process(or) executes a portable instruction set tailored to the C programming language. System calls and interrupt handling are built into the instruction set and architecture of the VPC process(or). The system calls allow programs to access the local and remote FIFOs and the Virtual File System (VFS). Timing facilities are also built into the processor so that simple real-time applications can be implemented.

The VPC process(or) can be *completely* controlled via the 5-bit remote control channel. The host can start and stop the VPC process(or) and can read and write memory. In a typical situation the host downloads a small bootstrap program into the VPC process(or) memory space. The VPC process(or) is then told to execute this bootstrap program, which reads a larger bootstrap from the 16-bit data channel. The host sends the program over the 16-bit data channel. When the program has been sent to the VPC process(or), the host requests execution using the 5-bit remote control channel.

The host can also load a small program into the VPC that causes a file from the VFS to be loaded into the RAM and executed. This avoids the problem of lengthy download times from the host.

The 5-bit remote control channel can be used by the host to control the VPC process(or). In some respects the 5-bit control channel can be viewed as a remote front panel for the VPC process(or). The 5-bit values (sometimes called quibbles for 5-bit nibbles) are divided into control codes and data codes.

If the upper bit of the 5-bit value is a one, then the low four bits contain data. An internal 16-bit register (called the Temporary or T register) is loaded with the 4-bit values when they appear on the channel. The low four bits of a 16-bit word are sent first, followed by the next four, and so on. If more than four data values (16 bits) are sent, the extra data are dropped. This allows for upward compatibility with the 32-bit version of the VPC.

If the upper bit of the quibble is a zero, then the low four bits contain a control code. The control codes can be used to start and stop the VPC process(or), read and write the 64K of memory, and read and write the VPC process(or) registers. The 16 control codes are shown in Table I (page 37).

Two internal registers are used in conjunction with the remote control channel. The T register is a holding register for quibbles sent from the host; its operation was described above. The P (or Pointer) register is used as an auto-incrementing pointer to read and write memory. The T and P registers are *not* accessible by an application program running on the VPC process(or).

The typical sequence used to load data into memory or a register is to send the four quibbles of data followed by the opcode. The T register is zeroed *after* an opcode is executed, and any quibbles that follow are loaded starting in the low-order bits as previously described. To get everything in synch, one usually starts by sending a Write P Register opcode first. This gets the T register ready to accept quibbles. The data is then sent, followed by another Write P Register opcode, which loads the data just sent into the P register.

At this point the T register is again reset to zero, and data can be loaded using quibble codes. A Write Memory opcode is then sent, causing the contents of the T register to be written into the memory location(s) pointed to by the P register. The P register is auto-incremented by the number of bytes in the word of the VPC (currently two for 16-bit VPC). The contents of the T register are written into memory starting with the least significant byte.

When an opcode is sent that requests a read operation, the reverse action occurs. The contents of the desired source are sent, four bits at a time, followed by the opcode that was used to request the read. In other words, if a Read Program Counter opcode is sent, the contents of the program counter will be sent to the host followed by a Read Program Counter opcode.

When a Halt or Run opcode is received by the VPC process(or), the proper action is taken and the respective opcode is returned to the host as an acknowledgment of the request.

The VPC registers and memory can be read (and written) while the VPC is executing programs. The returned values represent a snapshot taken at the time the request reaches the VPC. If the registers or memory is changed while the VPC is running, unpredictable results can occur. The operations are well behaved, but no synchronization with the currently running program is guaranteed.

Programs that run in the VPC process(or) have complete control over the machine. Data can be exchanged with the host or peripherals without host intervention. Sophisticated, user friendly, front end processors can be developed, freeing the host from such time-consuming tasks. Because the VPC process(or) is totally dedicated to this single task, no delays are experienced by the user. The user has a dedicated personal computer connected to the host in an integrated manner. The host can utilize as much or as little of the power of the VPC process(or) as it desires.

The User Layer

The user layer is used to interface the VPC process(or) to the peripherals that are available to the user. A variety of peripheral combinations can be developed for various applications. The common peripherals are a video display, a keyboard, an audio device, a graphics display, and a serial printer. All of the peripherals can be controlled by the VPC process(or) through the local FIFO.

The local FIFO is similar to the remote FIFO. The VPC process(or) can read and write to the local FIFO at any time. Data written into the local FIFO is stored and eventually processed by the Peripheral Interface Unit (PIU). The PIU decodes the upper eight bits of the 16-bit word written into the local FIFO as a channel number. The lower eight bits are sent to the device (or device driver software) connected to the specified channel. By convention the lower bit of the upper eight bits is used to indicate control or data. Each device can be controlled independently of the data flowing between the device and the VPC process(or). When the bit is a zero a data path is indicated, and when the bit is a one a control bit is indicated.

This arrangement allows up to 128 devices (or drivers) to be connected to the PIU. Each device has a control and a data path. The control and data paths are both bidirectional. Communication between the peripherals and the PIU is completely asynchronous. No master-slave relationships exist except by local conventions established for specific applications.

The information exchanged is maintained in a FIFO queue. The queue can hold 256 16-bit words for each direction. Flow control is the responsibility of the VPC process(or) and the peripherals (or drivers). Peripherals should *not* allow the FIFO to fill as a means of indicating a "not ready" condition; the control channel should be used for this purpose. This prevents the system from completely locking up or blocking because a peripheral is not ready.

Three virtual peripherals or device drivers have been proposed initially for the VPC. The Virtual KeYboard (VKY) device is used for all key entry and supports a typical ASCII type of keyboard with function keys and special editing keys. The Virtual DiSplay (VDS) device is an 80 by 24 monochrome device with a simple window-oriented protocol. The Virtual AuDio (VAD) device is an 8-bit sound generator used to create audible tones and simple music.

The peripherals are connected to the PIU using channels 0 (VKY), 1 (VDS), and 2 (VAD). Note that the channel number is encoded in the upper seven bits of the local FIFO word. The low bit selects control or data as noted earlier. Therefore, the VKY device actually uses binary channel 0 (data) and 1 (control). The

VDS uses 2 and 3, and VAD uses 4 and 5.

In the future more peripherals will be specified. Channel 3 has been reserved for the Virtual PRinter (VRP) driver, and channel 4 has been reserved for the Virtual GRaphics (VGR) driver. The VGR driver supports a subset of NAPLPS.

Even though any 8-bit character-oriented device can be connected to the local FIFO, a small number of compatible devices will be specified for anyone interested in working on a standard configuration. The following sections provide information on the three primary devices: VDS, VKY, and VAD.

VDS: The Virtual Display Device

The VDS device supports a simple window-oriented display compatible with 80 by 24 terminals. Partial screen scrolling is supported as well as business graphics for drawing simple forms.

The VDS protocol consists of 8-bit values sent to the VDS device in a stream format. The 8-bit bytes can be divided into the four groups shown in Table II (page 37). The VDS protocol can be implemented on most popular terminals and personal computers. Figure 3 (below) illustrates the control codes, window graphics, and ASCII characters supported by the VDS device.

0000	Halt the VPC Process(or)
0001	Run Starting at Location in PC
0010	Read Memory *P++
0011	Write Memory (*P++=T)
0100	Read P Register
0101	Write P Register (i.e., P=T)
0110	Read Program Counter
0111	Write Program Counter
1000	Read Stack Pointer
1001	Write Stack Pointer
1010	Read Frame Pointer
1011	Write Frame Pointer
1100	Read Argument Pointer
1101	Write Argument Pointer
1110	Read Literal Pointer
1111	Write Literal Pointer

Table I
Remote-Control Control Codes

0000xxxx	Control codes
0001xxxx	Window graphics
0xxxxxxx	ASCII characters
1xxxxxxx	Set buffer address

Table II
VDS Code Groups

VDS Control Codes

The 16 VDS control codes are listed in Table III (right). Most of the control codes are self explanatory. One unique feature is the concept of marks. A window is defined by two marks that determine a unique rectangular area on the screen. The marks are set by positioning the screen address to a specific place and issuing a Set Mark 1 or Set Mark 2 code. Once the marks are set, the various window operations apply to the area bounded by the rectangle determined by the marks. In other words, to clear a portion of the screen, one sets up an area and issues a Clear Window code.

VDS Window Graphics

Window graphics are used to create forms, tables, windows, and so on. The 16 window graphics codes have the following basic format:

0001xxxx

The four low bits are used to independently set one of four segments.

Five of the window codes are not generally useful. The code 0000 would produce a blank character, which would be redundant with the ASCII space character (octal 040). The window graphics code 0000 should be displayed as a centered dot. This character can be used to indicate an unused space in word processing applications.

000	CLW	CLear Window
001	CEL	Clear to End of Line
002	SM1	Set Mark 1
003	SM2	Set Mark 2
004	SWU	Scroll Window Up
005	SWD	Scroll Window Down
006	SWL	Scroll Window Left
007	SWR	Scroll Window Right
010	CON	Cursor ON
011	COF	Cursor OFf
012	HON	Highlight ON
013	HOF	Highlight OFf
014	ILI	Insert LIne
015	DLI	Delete LIne
016	ICH	Insert CHaracter
017	DCH	Delete CHaracter

Table III
VDS Control Codes

The codes 0001, 0010, 0100, and 1000 have only one segment lit and are not generally used in window and form drawing. These codes are displayed as arrows oriented so that the tip of the arrow exits the side of the cell that would normally have a lit segment. The code 0001 is displayed as an Up arrow.

The complete set of business graphics codes is listed in Table IV (page 38).

VDS Set Buffer Address

The buffer address (i.e., cursor position) is set using codes from 128 to 255. Direct and relative positioning is provided. Direct positioning is performed in relation to the physical display screen. Relative positioning is performed relative to the active window.

The codes 128–151 are used to set the relative address to the beginning of any of the 24 rows of the active window. Row 0 (i.e., code 128) is the first row of the window but not necessarily the physical screen. If a row is specified that is outside the active window, then a wrap-around effect occurs. The column will be set to the left-most edge of the active window.

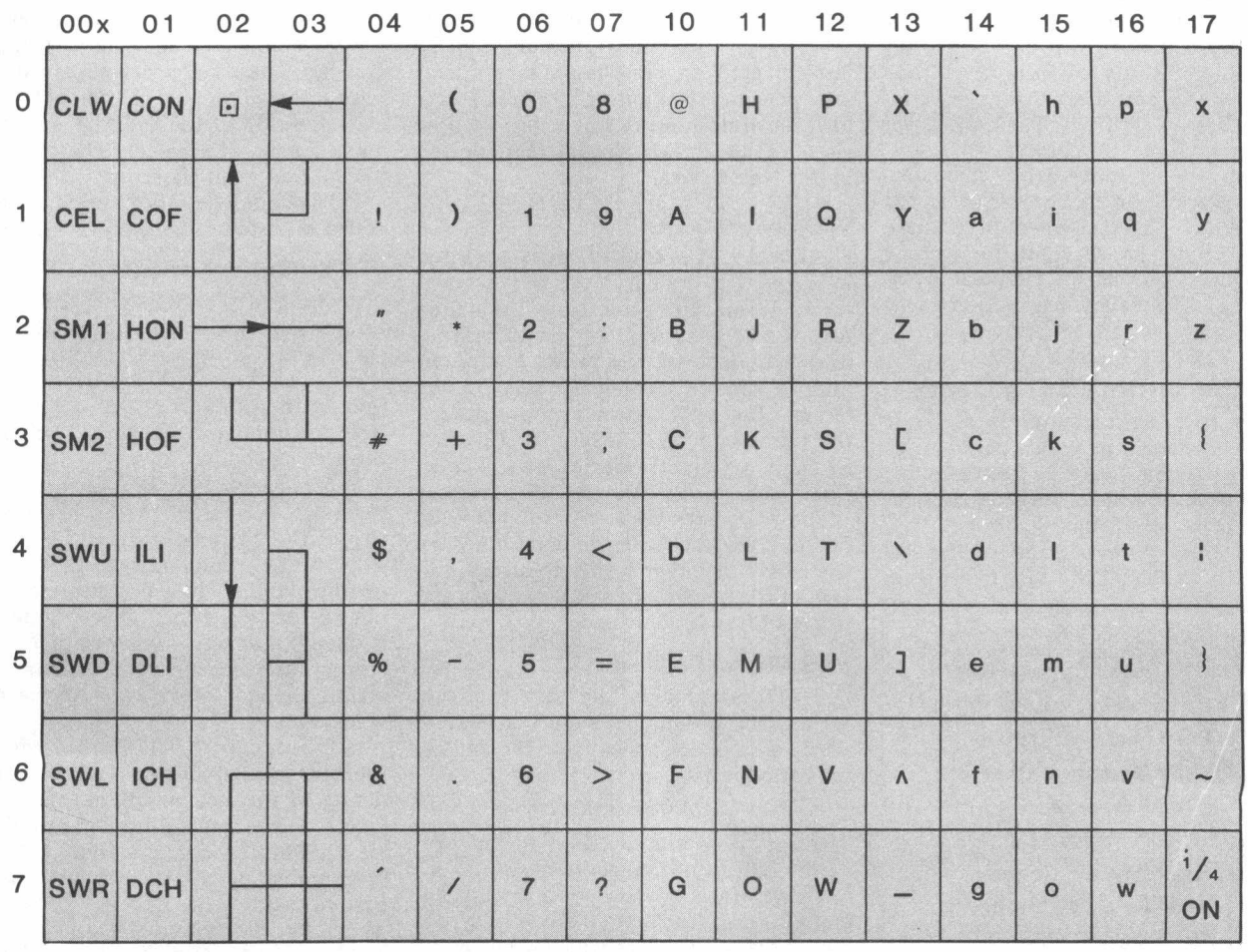
The codes 152–175 are used to set the buffer row address (0–23) independently of the current column and current window. The codes 176–255 are used to set the buffer column address (0–79) independently of the current row and current window.

VKY: The Virtual Keyboard Device

The virtual keyboard driver is an 8-bit device that produces 128 codes corresponding to the keys most commonly used in personal computer applications. As with other device drivers, the VKY has

0000	Centered dot
0001	Up arrow
0010	Right arrow
0011	Lower left corner
0100	Down arrow
0101	Vertical edge
0110	Upper left corner
0111	Left edge right tee
1000	Left arrow
1001	Lower right corner
1010	Horizontal edge
1011	Bottom tee
1100	Upper right corner
1101	Right edge left tee
1110	Top tee
1111	Crossing lines

Table IV
VDS Window Graphics Codes



	00x	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
0	CLW	CON	⊡	←		(	0	8	@	H	P	X	`	h	p	x
1	CEL	COF			!	)	1	9	A	I	Q	Y	a	i	q	y
2	SM1	HON	→		"	*	2	:	B	J	R	Z	b	j	r	z
3	SM2	HOF			#	+	3	;	C	K	S	[	c	k	s	{
4	SWU	ILI			$	,	4	<	D	L	T	\	d	l	t	¦
5	SWD	DLI			%	-	5	=	E	M	U	]	e	m	u	}
6	SWL	ICH			&	.	6	>	F	N	V	^	f	n	v	~
7	SWR	DCH			'	/	7	?	G	O	W	_	g	o	w	¼ ON

Figure 3.
VDS Codes

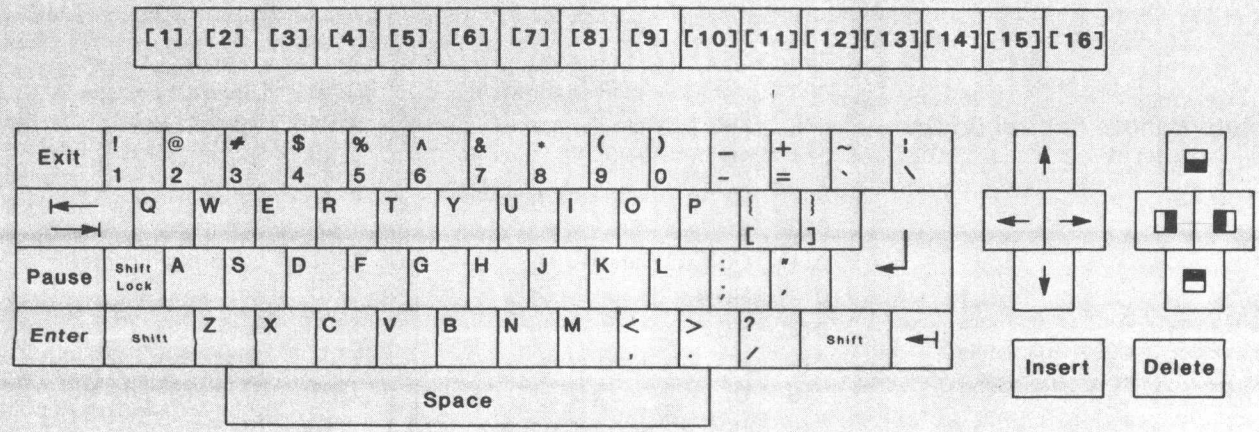


Figure 4.
VKY Functional Keyboard

000 - 003	Directional keys (arrows)
004 - 007	Page keys
010 - 017	Editing and control keys
020 - 037	Function/Menu keys ([1] to [16])
040 - 176	ASCII printable codes
177	Exit

Table V
VKY Code Groups

010	Backspace	(Destructive)
011	Tab	(Next field)
012	New line	(RETURN)
013	Back tab	(Previous field)
014	Pause	(Not automatic)
015	Enter	(Done, EOF, etc.)
016	Delete	(Character, Page, Window, etc.)
017	Insert	(see Delete)

Table VI
VKY Editing and Control Keys

000	Up	040	Space	100	@	140	`
001	Down	041	!	101	A	141	a
002	Left	042	"	102	B	142	b
003	Right	043	#	103	C	143	c
004	Begin	044	$	104	D	144	d
005	End	045	%	105	E	145	e
006	Previous	046	&	106	F	146	f
007	Next	047	'	107	G	147	g
010	Backspace	050	(	110	H	150	h
011	Tab	051	)	111	I	151	i
012	New line	052	*	112	J	152	j
013	Back tab	053	+	113	K	153	k
014	Pause	054	,	114	L	154	l
015	Enter	055	-	115	M	155	m
016	Delete	056	.	116	N	156	n
017	Insert	057	/	117	O	157	o
020	[1]	060	0	120	P	160	p
021	[2]	061	1	121	Q	161	q
022	[3]	062	2	122	R	162	r
023	[4]	063	3	123	S	163	s
024	[5]	064	4	124	T	164	t
025	[6]	065	5	125	U	165	u
026	[7]	066	6	126	V	166	v
027	[8]	067	7	127	W	167	w
030	[9]	070	8	130	X	170	x
031	[10]	071	9	131	Y	171	y
032	[11]	072	:	132	Z	172	z
033	[12]	073	;	133	[	173	{
034	[13]	074	<	134	\	174	|
035	[14]	075	=	135	]	175	}
036	[15]	076	>	136	^	176	~
037	[16]	077	?	137	_	177	Exit

Table VII
VKY Key-codes

been designed to support very primitive keyboards by mapping common control keys to the VKY keys, yet it can support the exotic keyboards found on today's personal computers with a direct mapping of keys. A balance has been struck between the new and the old.

Figure 4 (below) illustrates the VKY (although this keyboard does not exist physically). Users should become familiar with the mapping of their physical keyboards to this functional set. A typical dumb terminal used with the Unix operating system usually maps the PAUSE to CTL-S, ENTER to CTL-D, and EXIT to CTL-C or DEL.

The VKY device has been designed to avoid having too many keys available for applications. Originally the intent was to develop a 256-key generic keyboard that would support everything but the kitchen sink. Several common keyboards were considered (i.e., IBM-PC, DEC-VT 100, DEC-PC, EPSON QX-10, etc.) from which a rather impressive set of keys can be assembled. But even though there are a considerable number of common keys (like A–Z), the lack of consistency was discouraging.

Several problems occur when the 256-key approach is used. The most serious problem is that the average user cannot remember 256 functional keys, much less the contortions that are involved to activate the keys. Another problem is that when 256 codes or functions are available, the application program must be more agile than when only 128 codes are used. Lastly, when 256 functional key-codes are specified, the mapping of those codes to the popular keyboards becomes a rather arbitrary and frustrating task. With such a large number of key-codes available, all judgment calls end up by supporting both keys even though they may be very close in function. For example, the keys PAGE UP (IBM-PC) and PREVious PAGE (DEC-PC) are obviously meant for the same purpose. When 256 codes are available, it is tempting to support both in case another computer arrives on the scene with both keys.

The alternate approach chosen for the VKY device was to limit the keyboard to 128 codes (seven bits). The problem with taking this approach is that standard ASCII codes fill the entire key table: the 95 visible character codes plus the 32 control codes plus DEL make up the 128 character set. It is obvious that the 95 visible characters should not be touched. The remaining 33 codes are more than adequate for function keys and editing keys, but the common ASCII control codes have to be eliminated. This is not a major penalty because many users are confused when confronted with cryptic CTL-xx input techniques.

In fact, it turns out that many of the

existing ASCII control codes can be preserved because of their widespread support on keyboards. New codes can be dropped in among the preserved codes, resulting in a very terse set of usable keys that almost any major personal computer can support in a manner that is obvious to a novice user. A side benefit of this approach is that those systems that have a primitive keyboard can use the CTL-xx combinations to generate the VKY codes because CTL-xx codes are not normally supported.

Another benefit of 7-bit (128) coding is that the high bit is now free for use as a KEY-SPEED bit. The VKY device sets the high order bit to zero when keys are struck in a normal manner. The high bit should be set to a one when a key is generated less than one tenth of a second after the previous key. The value in the 16-bit VTM timer is saved each time a key is generated. If another key is generated and the value in the VTM timer has not changed, the high bit of the new key is set to one.

The KEY-SPEED bit is normally set when a user is striking a single key at a rapid rate or when a special device (like a joystick) is being used to generate key strokes. Application programs can always ignore the high bit, the only possible penalty being read-time response. The presence of the high bit, however, can be used by the application program to indicate that it would be wise to raise the priority of keyboard processing since keys are being pressed at an unusually rapid rate. The functionality of each key should *not* be changed by the presence of the KEY-SPEED bit.

Keyboards and terminals with auto-repeat functions generate multiple keys when keys are continuously held. Keys are usually repeated 15 to 30 times per second. The high-order bit periodically will be set to one when auto-repeating keyboards are used. This is normal, and the application program can use this information to the best of its ability.

If special devices (like joysticks or touch pads) are used to generate multiple keys, then the KEY-SPEED bit should be set to indicate that a high-speed key input device is being used. Again, the KEY-SPEED bit should be used to indicate not a change in functionality but rather a change in key generation speed.

Various studies and large amounts of head holding have resulted in the following specification for the VKY device driver. One must keep in mind that the spirit of this effort is to identify functionality and not key-top identification. The VKY device keys can be grouped into the six groups shown in Table V (above).

VKY Directional Keys

The directional keys are the first four key-codes. The four directions (up, down, left, and right) are provided. Arrow keys are usually used to generate these codes. When diagonal movements are desired, a two-code sequence should be used with the KEY-SPEED bit set in the second key to indicate the composite action: if the application program makes use of the KEY-SPEED bit, it can interpret the two moves as a single move.

Users with only four arrow keys can achieve the same result by pushing two of the direction keys in rapid succession. Sophisticated keyboard detection schemes can also be used to allow a user to push two arrow keys at the same time. The resulting codes sent to the application program are the same.

If joysticks, mice, or touch discs are used, an appropriate stream of direction keys should be generated.

VKY Editing and Control Keys

This group of eight keys is probably the most controversial. This group has been formed as a result of careful selection with an eye toward the past, present, and future. The functionality of the keys may not be as controversial as the recommended physical keyboard equivalents. The functional codes are shown in Table VI (page 40).

VKY Function Keys ([1] to [16])

The codes 020–037 are very straightforward. They are used to encode special function keys that are used by an application in a specific manner. It is expected that these codes should be generated by physical keys that can be relabeled for the application. In units that have only 10 function keys (like the IBM-PC), it is anticipated that some CTL or ALT arrangement will be used to generate all 16 keys. When terminals like the VT-100 are used, the typical approach is to convert the numeric keypad into a function key cluster. When this approach is used, a two-key sequence can be used for each function key. The function keys [1] to [9] are actually generated by the sequences [0][1] to [0][9]; the function keys [10] to [16] can be generated by two-key sequences [1][0] to [1][6]. When the [0] key is pressed, the device waits for another key to make a final determination.

Another approach that can be used with a numeric keypad is to map the keys [1] to [9] to the function keys [1] to [9], with the [0] key treated as function [10]. The keys [11] to [16] are generated by the two-key sequences [1][1] to [1][6]. This approach requires that timing detect the difference between [1] and [1][X] combinations. This can be accomplished by starting a timer whenever the [1] key is pressed. If another key does not follow in a short period of time (maybe one second), the [1] function key is generated.

The latter approach to using the numeric keypad is usually more popular because the majority of the function keys can be generated by a single key-stroke. Other methods can be devised; it does not matter how the function keys are generated as long as the proper key-codes are generated for the application program.

	00		01		10		11	
0000	A	55.00	C#	138.59	F	349.23	A	880.00
0001	A#	58.27	D	146.83	F#	369.99	A#	932.32
0010	B	61.73	D#	155.56	G	392.00	B	987.76
0011	C	65.41	E	164.81	G#	415.00	C	1046.52
0100	C#	69.29	F	174.61	A	440.00	C#	1108.72
0101	D	73.42	F#	184.99	A#	466.16	D	1174.64
0110	D#	77.78	G	196.00	B	493.88	D#	1244.52
0111	E	82.41	G#	207.50	C	523.25	E	1318.52
1000	F	87.31	A	220.00	C#	554.36	F	1396.92
1001	F#	92.50	A#	233.08	D	587.33	F#	1479.96
1010	G	97.99	B	246.94	D#	622.26	G	1568.00
1011	G#	103.75	C	261.63	E	659.26	G#	1660.00
1100	A	110.00	C#	277.18	F	698.46	A	1760.00
1101	A#	116.54	D	293.66	F#	739.98	A#	1864.64
1110	B	123.47	D#	311.13	G	784.00	B	1975.52
1111	C	130.81	E	329.63	G#	830.00	C	2093.04

Table VIII
VAD Note Codes

VKY ASCII Key-codes

The codes 040–176 are equally straightforward. They are generated by the common ASCII punctuation and alphanumeric keys. The physical Shift key is typically used to access the upper-case and lower-case characters.

It should be noted that keys like Shift, Shift Lock, and Numeric Lock are all considered to be private to the keyboard and/or VKY device driver. The application program does not need to know precisely how the key was generated: it is concerned only with the intent of the user.

VKY Exit Key

The last bit of functionality needed in most applications is the Interrupt or Attention capability. The key-code 0177 is used to encode this function. The key has been named Exit to balance out the Enter key and some of the terminology from the system.

The physical DEL (Delete) key can be used for the Exit function although the ESC (Escape) key can be equally useful. Note that the normal ASCII ESC code 033 is used by the F12 special function key, which means the physical ESC key is now available. The CTL-C combination is another popular key used to generate the Exit key-code.

If the distinction between functional (logical) and physical keys is still confusing, the following example may help. If the ESC key is chosen for the Exit function, then each time the ESC key is pressed the VKY device should send the VPC processor an 0177 code indicating an Exit. The application program running in the VPC should *always* interpret the 0177 key-code as an Exit request whether or not it knows that the ESC key was used to generate the code. Alternate physical keys that can be used for Exit are the BREAK key or the DEL key. The DEL key should be used as a last resort because it is more commonly used to generate the Delete key-code (octal 016).

Each of the keys shown in Figure 4 generates a key-code. Table VII (p. 40). illustrates all of the key-codes supported by the VKY device. The key-codes should be handled by an application program as functional codes rather than as physical codes. The physical keys that are pressed do not have to be precisely labeled with the key-codes in the table. The user should be concerned with the desired function rather than the physical keys. This emphasis on functional keys allows application programs to be portable across a variety of machines.

VAD: The Virtual Audio Device

The Virtual Audio Device is an 8-bit device that allows a VPC program to generate simple sounds and music. Each 8-bit value written to the VAD device is used to immediately start a note of a particular frequency. The low six bits of the 8-bit byte are used to specify one of 64 notes. The upper two bits indicate the duration of the note.

The 64 notes correspond to the most common notes found in music. The notes range from A (note 0) two octaves below middle C to C (note 63) three octaves above middle C. Only the legal piano notes are supported. This arrangement allows six bits to represent tones from 55.0 Hz to 2953.0 Hz. The 64 notes are shown in Table VIII 9 (page 42) along with the frequency in Hz (cycles per second). The labels on the top of each row represent the high two bits of the note code.

The upper two bits of the 8-bit VAD code are used to control the duration of the tone. Table IX (at right) illustrates the four possible duration settings. The timing of the VAD tones is synchronized to the VTM timer. The VTM timer is a free-running 16-bit timer that is used by the VPC process(or) to schedule events and for other timing applications. The VTM timer ticks (i.e., increments) every one tenth of a second.

When an 8-bit byte is written to the VAD sound generator, the sound is instantly started. The note is played until one, two, three, or four ticks of the VTM timer occur. If a previous sound is still being played when a new value is loaded, the old sound stops and the new sound begins.

Note that if a sound is started very close to the next tick of the VTM timer and only one tick is requested then a short note may be heard. Normally, the VAD device is loaded as part of the VTM interrupt handler, which is executed at the beginning of a VTM time interval. This more or less guarantees that a full tenth of a second note will be generated. When two, three, or four ticks are requested, the notes will play for approximately two, three, or four tenths of a second, respectively.

The automatic turn-off feature of the VAD sound generator is extremely useful when a host computer wishes to send an isolated note to a terminal to alert the user. The host program can address the code to the VAD data channel, and the sound will be loaded, played, and ended without further codes being sent from the host.

00	Turn off on next tick
01	Turn off on tick after next
10	Turn off on third tick
11	Turn off on fourth tick

Table IX
VAD Duration Codes

The VAD device can be used to generate the common ASCII BEL code for backward compatibility with existing applications. The code for middle C (00011011) is normally used.

On systems without a general purpose sound generator, middle C is usually detected and converted to the BEL character on the terminal being used. When music is sent to a VPC, the occasional occurrence of middle C in a song will ring the BEL. The user may not be able to enjoy the song, but at least something is heard to indicate music is being played. The user is usually given the option to turn this feature off in the Set-up capability local to a VPC.

The Levels of VPC

At the present time three basic VPC configurations are defined. For a lack of better names, they have been designated level 0, level 1, and level 2.

The level 0 VPC is a terminal-only implementation. The local and remote FIFOs are patched together so that the host can communicate with all of the serial device drivers. The 5-bit control channel is not connected to anything and will not respond. No downloading of software can be performed. No files can be transferred.

The level 0 VPC can be used in applications that require a simple audio, video, and keyboard combination. Up to 128 serial devices can be handled using the standard FIFO driver interface techniques. Each key that is generated by the VKY is sent to the host. Likewise, all VDS screen control must come from the host.

A level 0 implementation of VPC has been developed by Unir Corporation for members of The Unir Project. It is available to members for $50.00 and includes a complete C language source code listing. Even though the package has been written for the industry standard VT-100 terminal, it can be easily adapted to other terminals.

Figure 5 (below) shows the modules included with the level 0 VPC package. The listings included with this article comprise the two rightmost elements (C78 and C87) and the two parts of the level 0 VPC (MUXPIU and PIUMUX). The relatively lengthy code for the VDS and VKY is not included. *[If readers show sufficient interest, we will run the code for those modules in a future issue. – Ed.]*

The level 1 VPC is a diskless version with all of the level 0 features plus the 64K RAM and VPC processor. Downloading can be done from the host using the 5-bit control channel in conjunction with the remote FIFO.

The level 1 VPC boots in the level 0 terminal mode. The local and remote FIFOs are patched together, and the VPC processor begins looping and checking the

two FIFOs. Data that appear on one FIFO are transferred into the other.

The level 2 VPC is equipped with a file system (VFS) for local file storage. Using the VPC processor, files can be stored and retrieved. A standard Unix-like hierarchical file system is assumed by the system calls.

The level 2 VPC boots from a file designated during the local setup phase of VPC. This file is loaded into the RAM (using the exec system call) at location 0 and given control. The standard default boot file brings the VPC up running the same program as level 0 and level 1.

The VPC level can be detected in a variety of manners. The first step is to attempt to read the program counter using the 5-bit control channel. If no response occurs, a level 0 VPC is being used. If a response comes back, a level 1 level 2 VPC is in use.

Further memory interrogation can be done to determine if a level 2 VPC is being used. As a convention, the level 2 VPC will have argc and argv set during the file loading process. The name of the file can be determined by interrogating the argument stack. In contrast, the level 1 VPC will not have the file name on the stack.

Testing Your VPC

Unir Corporation is providing a free data line for people interested in testing their VPC implementations. The data line features a member directory, a bulletin board service, a news service, and several VPC exercise programs.

A level 0 VPC is needed to access the system. A 7-bit asynchronous, even parity data link is used to simplify the access. The system can be reached 24 hours a day by dialing (317) 842-7014.

More information on VPC, the VPC data line, and The Unir Project can be obtained from Unir Corporation, 5987 E. 71st Street, Suite 106, Indianapolis, Indiana 46220.

Conclusion

The virtual personal computer is a general system for integrating timesharing and personal computer systems. The VPC replaces the common CRT and keyboard with a complete programmable workstation. The primary source of programming for the VPC comes from a host system and not the user.

The VPC can be used as a simple terminal or as a powerful component in a distributed processing system. Programs can be downloaded from a host into the VPC memory and can be executed as high-speed, real-time tasks. Files can be transferred between the VPC and the host without extra hardware or software. A wide variety of peripherals can be controlled by the VPC using a general purpose I/O capability. The user interfaces to the VPC and ultimately to the host, through these peripherals.

Unlike many software systems, a VPC makes no prior assumption about how these various capabilities are going to interact. Because of this, a VPC can be configured as a multi-screen terminal emulator, as a real-time front-end processor, or a simple personal computer. Concepts like windows and concurrent, multiplexed I/O are easily integrated into a system using a VPC. As the system evolves, the VPC can be reconfigured to meet the demands of new applications.

The VPC is extremely useful in bridging the gap between personal computers and large, time-shared systems. This gap represents one of the next major frontiers in the computer industry. ■■J

(Listing begins below)

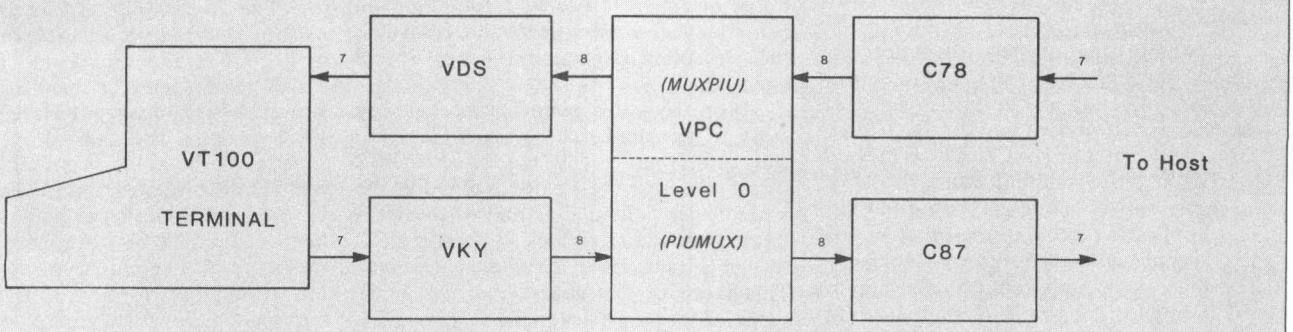


Figure 5.
Level 0 VPC

Virtual Personal Computer (Text begins on page 32)

Listing One

```
#include <stdio.h>

/*
*
*       convert 8 bit mux format (mux)
*       to 7 bit even parity format (c7e)
*
*/

/*
*
```

```
*       even parity table
*
*/

char parity[] = {
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80
};

int lbase = 0;          /*      last base               */


/*
*
*       8 bit characters are read that contain
*       8 bits of data
*       they are converted to 7 bit format and
*       even parity is set in the upper bit
*
*/


main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
        int c;


        setbuf(stdin,NULL);
        setbuf(stdout,NULL);
        new_win(0);
        while((c=getchar()) != EOF){
                put_c7e(c);
        }
}

/*
*
*       set up new window
*
*       the offset of 8 is not encoded
*       but is set for later use
```

(Continued on next page)

Virtual Personal Computer

Listing One

(Listing continued, text begins on page 32)

```
*
*/


new_win(byte)
int byte;
{
        lbase = ((byte-8) & 0xe0) + 8;
        put_par((lbase>>5) + 32);
}

/*
*
*       convert 8 bit bytes
*       to 7 bit format
*
*       7 bit format
*
*       3 bit base + 8      XXX01000
*       6.45 bit offset   + 0xXXXXXX
*                           ----------
*       8 bit result        XXXXXXXX
*
*       the 3 bit base is encoded
*       as numbers in the range
*       32 to 39
*
*       the extra offset of 8 is added
*       to provide more efficiency
*       when handling ASCII files
*
*       the extra offset of 8 is not
*       encoded but is assumed as
*       shown above
*
*       the 6.45 bit offset is encoded as
*       numbers in the range 40 to 127
*
*       the 3 bit base is only changed
*       when necessary
*
*       if consecutive 8 bit numbers
*       fall in the same range then the
*       3 bit base is "shared" and
*       each 8 bit byte will only require
*       1 byte containing a 6.45 bit offset
*
*       the codes 0 to 31 are never
        enountered
*       so that control codes can be used
*       for flow control and other purposes
*
*       the codes 128 to 255 are not used
*       the high order bit is available for
*       a parity bit if needed
*
*/
```

```
put_c7e(byte)
int byte;
{
        if(((byte-lbase)&0xff) > 87){
                new_win(byte);
        }
        put_off(byte-lbase);
}

/*
*
*       output 6.45 bit offset
*
*       the offset should be in the range
*       0 to 87
*
*/

put_off(offset)
int offset;
{
        put_par(offset + 40);
}

/*
*
*       output 7 bits with even parity
*
*/

put_par(c)
int c;
{
        c &= 0x7f;
        putchar(c | parity[c]);
}
```

End Listing One

Listing Two

```
#include <stdio.h>
#include <sgtty.h>
#include <signal.h>

/*
*
*       convert 7 bit even parity format (c7e)
*       to 8 bit mux format
*
*/

int ttyflg = 0;

struct sgttyb savtty;
struct sgttyb newtty;

/*
*
*       even parity table
*
*/
```

(Continued on next page)

(Listing continued, text begins on page 32)

```
char parity[] = {
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x80,0x00,0x00,0x80,0x00,0x80,0x80,0x00,
        0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x80
};


/*
*
*       8 bit characters are read
*       the lower 7 bits contain data and the upper
*       bit has even parity set (hopefully)
*
*       parity is tested and if an error occurs
*       the character is dropped
*
*/

main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
        int c;
        int data;
        int resexit();

        init();
        signal(SIGINT,&resexit);
        signal(SIGHUP,&resexit);
        signal(SIGPIPE,&resexit);
        while((c=getchar()) != EOF){
                data = c & 0x7f;
                if(parity[data] == ((char)(c&0x80))){
                        put_mux(data);
                }
                else{
                        fprintf(stderr,"%s: parity error %o\n",argv[0],c);
                }
        }
        restore();
}

/*
*
*       init the tty if needed
```

```
*
*/
init()
{
        if(isatty(0) == 1){
            ttyflg = 1;
            gtty(0,&savtty);
            gtty(0,&newtty);
            /*      echo implicitly off  */
            newtty.sg_flags |= RAW;
            newtty.sg_flags &= ~ECHO;
            stty(0,&newtty);
            setbuf(stdin,NULL);
            setbuf(stdout,NULL);
        }
}

/*
*
*       send to mux
*
*       convert 7 bit bytes
*       to 8 bit format
*
*       7 bit format
*
*       3 bit base + 8    XXX01000
*       6.45 bit offset + 0xXXXXXX
*                         ----------
*       8 bit result      XXXXXXXX
*
*       the 3 bit base is encoded
*       as numbers in the range
*       32 to 39
*
*       the extra offset of 8 is not
*       encoded but is assumed as
*       shown above
*
*       the 6.45 bit offset is encoded as
*       numbers in the range 40 to 127
*
*       the 3 bit base is only changed
*       when necessary
*
*       if consecutive 8 bit numbers
*       fall in the same range then the
*       3 bit base is "shared" and
*       each 8 bit byte will only require
*       1 byte containing a 6.45 bit offset
*
*       the codes 0 to 31 are never enountered
*       so that control codes can be used
*       for flow control and other purposes
*
*       the codes 128 to 255 are not used
*       the high order bit is available for
```

(Continued on next page)

Virtual Personal Computer

Listing Two

(Listing continued, text begins on page 32)

```
*       a parity bit if needed
*
*/

put_mux(c)
int c;
{
        static int lbase = 0;
        char byte;

        byte = c & 0x7f;
        if(byte >= 32){
                if(byte < 40){
                        lbase = ((byte-32) << 5) + 8;
                }
                else{
                        putchar((byte-40)+lbase);
                }
        }
}

/*
*
*       restore and exit
*
*/

resexit()
{
        restore();
        exit();
}

/*
*
*       restore the tty
*
*/

restore()
{
        if(ttyflg == 1){
                stty(0,&savtty);
        }
}
```

End Listing Two

Listing Three

```
#include <stdio.h>

/*
*
*       multiplexor to piu conversion
*       level 0
*
*       an 8 bit stream is read from the standard input
*       and a 16 bit stream is written on the standard output
*
*       the 8 bit stream usually comes from the c78 converter or
*       other link level processors
*
*       the 16 bit stream that is written on the standard output
*       is created by writing 2 bytes
*       the low byte is written first followed by the high byte
*
*       the 16 bit channel is usually sent to a PIU or VDS module
*
*/

unsigned base = 0;

/*
*
*       mux input
*
*       break down as follows:
*
*               0XXXXXXX - add base to XXXXXXX yielding 16 bit data
*               10XXXXXX - set middle 6 bits of base
*               110XXXXX - set upper 5 bits of base
*               111XXXXX - 5 bit remote control channel value
*
*       the 5 bit remote control channel is dumped because this
*       is a level 0 VPC utility
*
*/

main()
{
        int c;
        int data;

        setbuf(stdin,NULL);
        setbuf(stdout,NULL);
        while((c=getchar()) != EOF){
                if(c < 128){
                        data = base + (c&0x7f);
                        put_piu(data);
                }
                else{
                        if(c < 192){
                                base &= 0xf81f;
                                base |= (c&0x3f) << 5;
                        }
                        else{
                                if(c < 224){
```

(Continued on next page)

Virtual Personal Computer (Listing continued, text begins on page 32)

Listing Three

```
                                        base &= 0x07ff;
                                        base |= (c&0x1f) << 11;
                                }
                                else{
                                        /*      eat the control channel */
                                }
                        }
                }
        }
}

/*
*
*       output to piu
*
*/

put_piu(data)
int data;
{
        putchar(data & 0xff);
        putchar((data>> 8)&0xff);
}
```

End Listing Three

Listing Four

```
#include <stdio.h>

/*
*
*       piu to multiplexor converter
*       level 0
*
*       this program reads the standard input and converts a 16 bit
*       stream to multiplexor format
*       the standard input contains 16 bit values encoded as 2
*       bytes, low byte followed by high byte
*
*       the standard output is used and contains data in 8 bit
*       multiplexor format
*       the output is usually directed to a link level process
*
*/

unsigned base = 0;

main()
{
        int low;
        int high;

        setbuf(stdin,NULL);
        setbuf(stdout,NULL);
        while((low=getchar()) != EOF){
                high = getchar();
                put_mux(((high & 0xff) << 8) + (low & 0xff));
        }
}

/*
```

```
*
*       send 16 bit data value through mux
*
*       there are other clever ways of setting the base values
*       to yield better link performance
*       this method is easy to program and describe
*
*       if the value to be sent can be reached using the current
*       base then only a 7 bit offset is sent
*       if the value can not be reached, a new base is
*       determined by the upper 11 bits of the current value
*       the 5 and 6 bit sub-base registers are compared with the
*       current base register and one or both are changed as needed
*       the value is then sent using this new base
*
*/

put_mux(value)
unsigned value;
{
        unsigned diff;
        unsigned new_bas;

        if((diff = (value - base)) < 128){
                putchar(diff);
        }
        else{
                new_bas = value & 0xffe0;
                if((new_bas & 0xf800) != (base & 0xf800)){
                        putchar(((new_bas & 0xf800) >> 11) | 0xc0);
                }
                if((new_bas & 0x07e0) != (base & 0x07e0)){
                        putchar(((new_bas & 0x07e0) >> 5) | 0x80);
                }
                putchar(value & 0x1f);
                base = new_bas;
        }
}
```

End Listing Four

PABX and the Personal Computer

You are probably wondering why a journal like *Dr. Dobb's* is discussing phone systems. Well, the private area branch exchange (PABX) is now a prime candidate for use as a local area network (LAN) for computer connections. Let's take a look at what a PABX is and see if the PABX is really the LAN of the future or simply a way to get rid of some wire in a building.

What's a PABX?

The PABX is a local telephone system usually limited to a building, office, campus, or geographical area. Its purpose is to replace the "expensive" facilities of the telephone company with equipment owned or leased by the business itself to save money and improve communications. PABX systems are provided by common carriers such as the local Bell Telephone system or by independent suppliers such as Northern Telecom, Intecom, and others. Offices acquired PABXes because of a need to communicate within an organization without sending all calls through the local telephone system's central office switching facility.

State-of-the-art PABX systems have a number of distinguishing features that make them prime candidates for data movement media. One such feature is the digital nature of the switching facility. When you pick up the receiver on your digital telephone and "dial" a number, a connection is made at the PABX by routing the call through a digital switching circuit, usually via a time-division multiplexing matrix switch. When you talk, your voice is converted from an analog signal into a digital signal at a voice transmission rate of 32K or 64K bits per second. If the connection is within the building or switching area of the PABX, the signal remains digital. If the connection is to an outside facility, i.e., for a long distance call, the digital voice signal is again converted to an analog signal and placed on the common carrier trunk (the wires of the phone company).

Your digital phone set is an electronic marvel (see Figure 1, at right). It has a microprocessor built in to keep track of your speed-dialing numbers, the last number you dialed, and other features that are found on the friendly faceplate. It also controls the A/D conversion of your voice. Connecting your phone to the PABX are two pairs of wires: a power pair and a signal pair. The signal pair is used to transfer voice information and, in most cases, digital data traffic as well.

The Computer Connection

Every communications network has a topology, that is, the way the wires are connected together. The PABX uses a star network, one where stations are connected in radials from a central point of control (see Figure 2 on page 61). The first telephone system was one of the first star-configured communications networks. Then the mainframe computer came along and adopted the same structure for its computer-to-terminal connections. In fact, one of the first uses of a PABX as a LAN was in a mainframe/terminal interconnect. Why is the PABX a better solution than those available in the past?

For one thing, utilizing the PABX as an interconnect device gets rid of all of the twisted-pair wire that is usually strung around the building for data traffic. Simply add an RS-232 or RS-449 connector to the back of the phone set, include more logic in the phone and the switch used to handle the data traffic, and connect your terminal. Now you can use your phone to dial a computer connection. The PABX is also connected to the computer terminal port so that a "conversation" can take place between the terminal and the computer (see Figure 3, page 62).

Using the PABX also reduces the number of terminal ports required on the mainframe or minicomputer. Computer users are like phone users: not all of them use the computer all day. In a large office a limited number of outside telephone lines are usually shared by all of the workers. Those people that need to have regular outside access usually have a dedicated phone line for themselves. This keeps down the cost of the telephone service. The same applies to computer connections. In most installations casual users come and go; their hard-wired connections are wasted for a given part of the day (see Figure 4 on page 62). By connecting a certain number of terminal ports for casual users through the PABX, you can reduce the port configuration (see Figure 5 on page 63). When the user dials the computer, the PABX looks for the next available port and makes the connection. When a terminal is disconnected, a port is freed for other users to access.

The Micro Connection

LANs typically move data from one point to another using some common communications capability. The network allows the data to be looked at as either complete files that are moved around from workstation to workstation for pro-

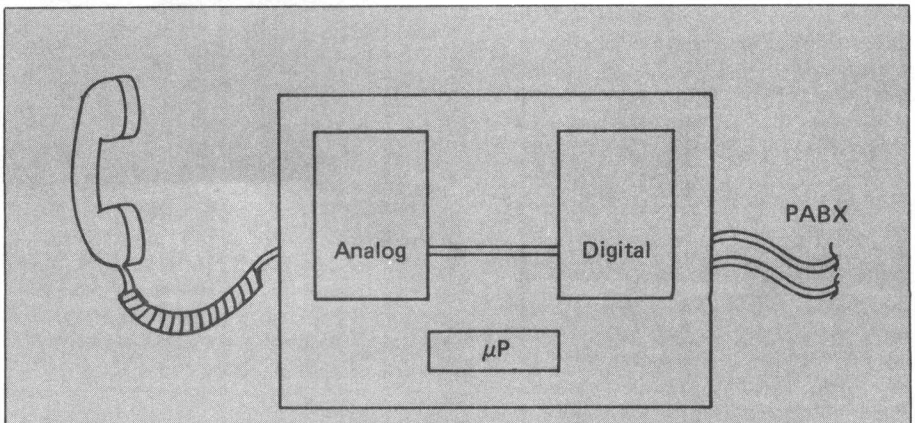


Figure 1.
Microprocessor Controlled Digital Phone

by Ted Rohling

Ted Rohling, 4041 Medical Drive, San Antonio, TX 78229.

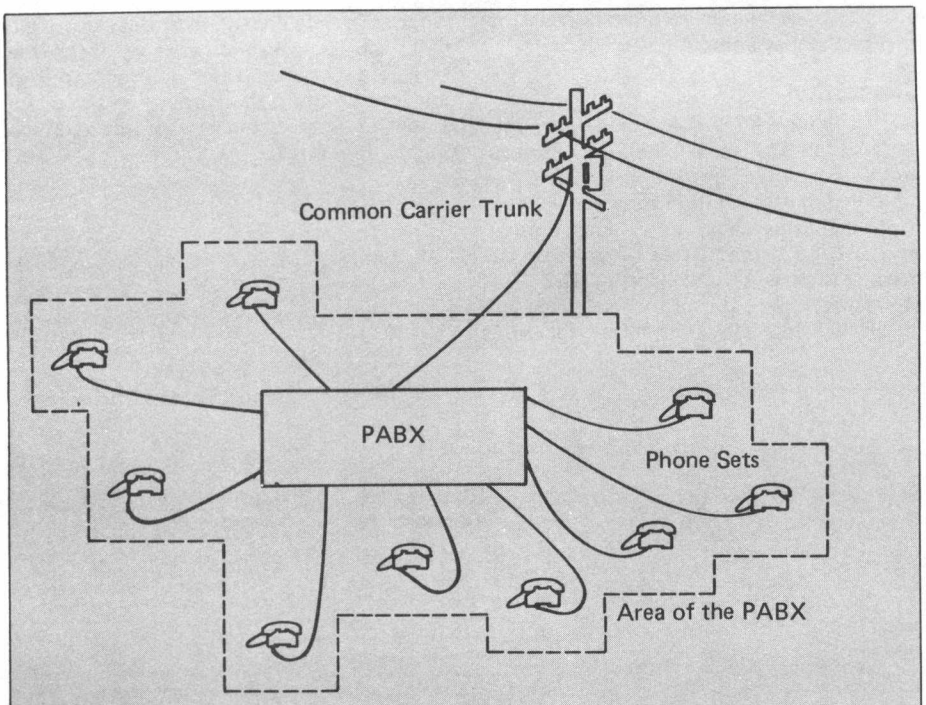


Figure 2.

PABX as a STAR Network

cessing or as "phantom disk drives." When using a LAN with phantom drives, you don't have to move the complete file to your workstation to process it. You can use the file where it is located, retrieving and updating records as if they were on your local disk devices. Certain networks utilizing coaxial cable and high-speed links can provide phantom drives.

Microcomputers in a PABX LAN installation are normally used in a file transfer or an attached terminal mode. The connections between micros within the scope of a PABX are simplified when a digital switch is used. The modems are replaced by the digital phones, and the phone connection is made as usual. The advantage is that a micro can now be moved from location to location without dragging the modem around with it.

Speed also is improved between 800% and 1600%, depending on the PABX utilized: data rates of 9.6 kilobaud or 19.2 kilobaud are common, which is a big improvement over the usual 300- or 12-baud modem used in dialup communications today. Moving a 20 kilobyte file at 300 baud under ideal circumstances takes over 11 minutes; at 9600 baud it takes about 21 seconds. These times obviously discard any disk I/O time or line error problems.

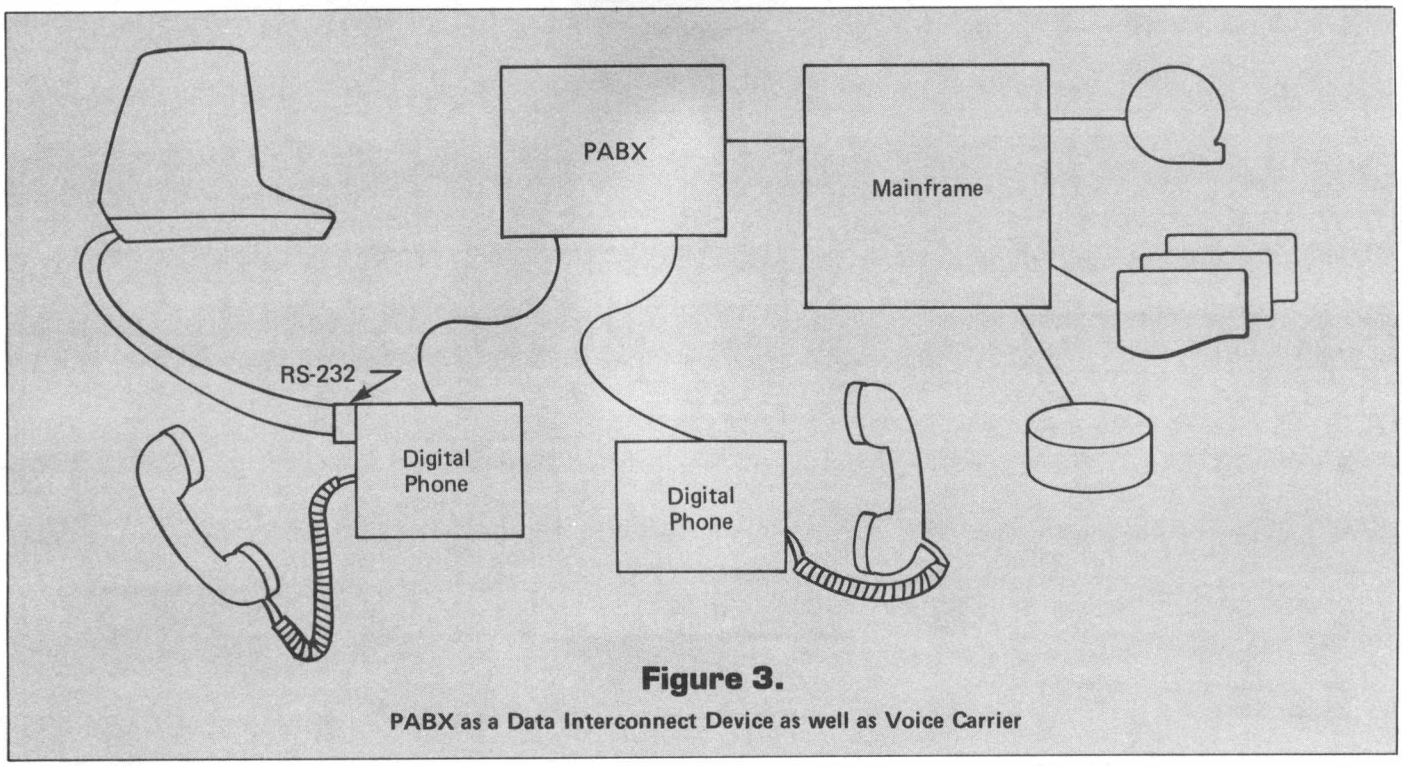


Figure 3.

PABX as a Data Interconnect Device as well as Voice Carrier

Is the PABX really a cost-savings approach to the local area networks? If a new installation is being developed, the answer is yes. Since today's average cost per telephone station for a digital PABX installation is $1000, the addition of the digital data capability costs about the same as a low-cost medium-speed modem. However, to completely retrofit an existing PABX with digital data capabilities simply for the LAN facility is not an economically feasible solution. Nor is the PABX a solution for small businesses, compared to the relatively low cost of "key systems."

Conclusion

Utilizing the PABX for a local area network in the microcomputer environment is not an optimum solution for most installations. Other network capabilities, including those now being defined by IEEE 802 committees or those on the market such as ARCNET or ETHERNET, are more applicable to the microcomputer world. However, for locations where digital PABX facilities are available, the users can now enjoy faster file transfers between their microcomputers and can connect their micro to the mainframe or minicomputer systems at much higher data rates. DDJ

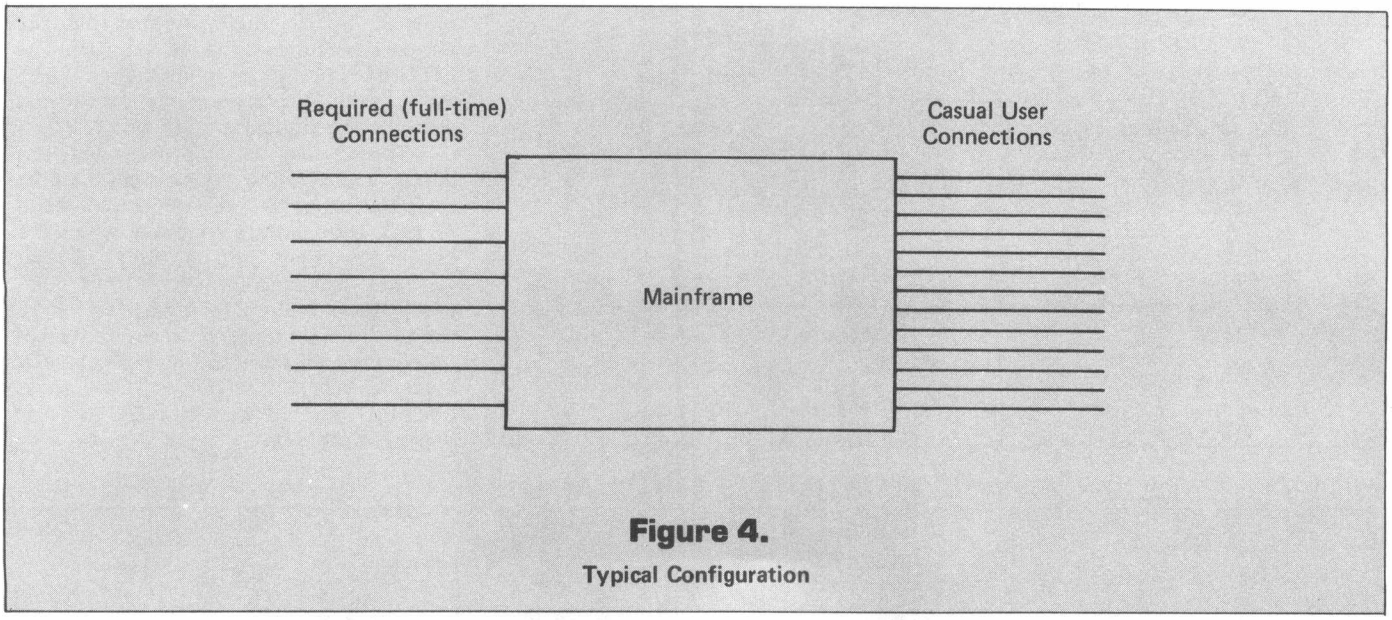


Figure 4.
Typical Configuration

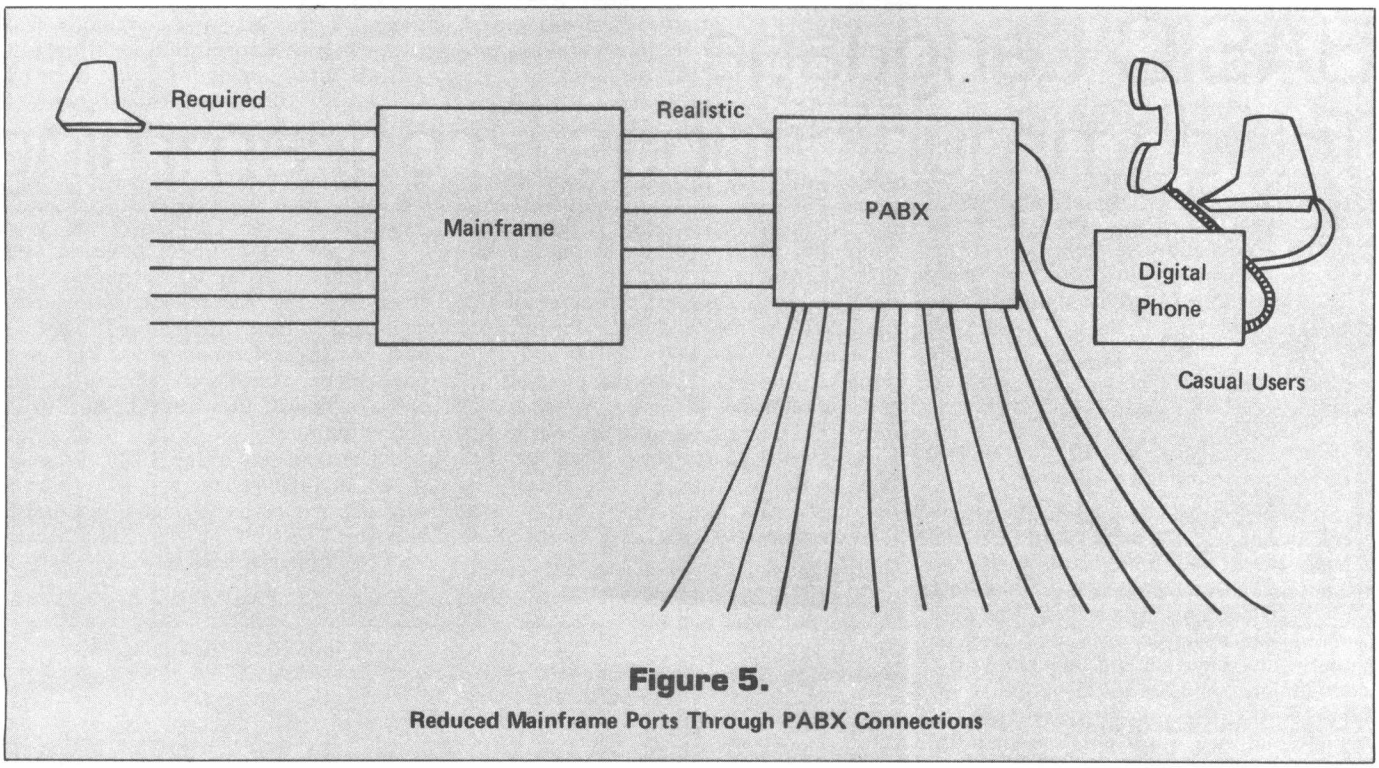


Figure 5.
Reduced Mainframe Ports Through PABX Connections

BASIC Language Telecommunications Programming

This article addresses some aspects of the following issues involving communications to a remote host (mainframe) computer:

- How to upload files to systems that will not let us install receive programs such as Modem7 protocol programs
- How to get BASIC to read the status of our microcomputer's hardware

The programs described in this article were developed to work with systems having these two limitations. First, we cannot install any programs on the host to help us communicate with it. The host is used, for example, as a mail drop or bulletin board system and only allows the computer to transfer ASCII text files; no binary format files may be transferred.

Second, we must not overload the host computer. Just about every mainframe computer that has keyboard or teletype (TTY) ports expects data to be transferred into that port at the typing speed of a human being. We, however, want to force-feed data into such a host's TTY port from our microcomputer; as a result, data will be lost when we overrun the ability of the host computer's operating system to absorb what we send. This sounds unbelievable, but it has occurred on every one of the dozen or so different mainframes we have tried this on. Some of these famous brands could not even accept data at a continuous rate of 300 baud (30 characters per second).

Echo Handshaking

The cure for all this is to sense the presence of an echo character coming back from the host and to use that to determine the timing of the whole system. This is called echo handshaking. This method is not limited by the baud rate of the modem being used: the transmission and echo times dominate the transmission speed.

The sequence goes something like this. Using some software means to directly connect our keyboard with the host through the printer/modem port, we prepare the host computer to accept data and to transfer it to a file in the host computer. We then start the echo handshaking communication program. The program sends the first character of the file up to the host and waits for the host to echo the character in normal full-duplex operation. When the program detects a returned character, it sends the next one, and so on until the entire file is transferred to the host. When the program is finished, we connect directly to the host again and terminate the transmission by whatever means the host wants.

We use the TELNET program, available on older BDS 'C' distribution diskettes, for the direct connection from keyboard to host through the CP/M LST: port on the microcomputer. This well-done piece of software allows the easy downloading of files from the host to the microcomputer.

The concepts in the BASIC programs shown in Listings One and Two (page 66 and 67, respectively) could be inserted into the TELNET program, but because we communicate with a wide variety of

by R. S. Broughton

Robert S. Broughton, P.O. Box 5191, Beaverton, OR 97006.

I/O READ PORTS:

	DATA	STATUS
Channel A, printer (LST:) port	80h	81h
Channel B, terminal (CON:) port	82h	83h

Figure 1.
I/O Register Addresses

Z80–SIO Receive Port Status Register:

D7	D6	D5	D4	D3	D2	D1	D0

Where the individual bits are defined as:

D7 – Break Abort
D6 – Transmit Underrun/EOM
D5 – Clear To Send
D4 – Sync/Hunt
D3 – Data Carrier Detect
D2 – Transmit Buffer Empty
D1 – Interrupt Pending
D0 – Receive Character Available

Figure 2.
Z80–SIO Status Register Description

host computers, we felt that a separate program for each protocol would suffice. We use these programs on a Cal-Tex BigBoard II microcomputer.

The program shown in Listing One will communicate with most Unix computer systems and bulletin board systems. For other computer hosts such as Hewlett-Packard (H-P) computers, however, a slightly different protocol is used. A program that uses this different protocol is shown in Listing Two.

The H-P computers allow data to be entered by using one of several editor programs (e.g., EDIT2 or EDITOR) after placing the program into insert mode. The protocol for these editors is slightly more complex than editors typical of the Unix systems. When in insert mode, the H-P editors send out a prompt for each line, which includes the current line number. This prompt ends with a control-Q (X-ON), which is the signal to begin entering the next line. After sending the carriage return for the line, the program in Listing Two discards characters received from the H-P host until the control-Q is received. The characters discarded are the prompt for the next line. The next line is then sent, character by character, each time waiting for the echo of the preceding character until the next character is sent, and so on.

Discussion

Since the programs are very similar, we will examine only the version for Unix computers in Listing One. The comments within the version for H-P computers in Listing Two should explain the protocol differences between the two. The statements are listed in Table I (below).

Statement 450 reads a byte from register 129 (decimal) or 81 (hex) by using the INP instruction. The AND 1 instruction masks or turns off all bits in this byte except for the low-order bit; on the BigBoard II microcomputer using the Zilog Z80-SIO serial I/O chip, this bit is the data-ready bit in the status register of the SIO. Statement 450 will loop on itself as long as this bit is a zero, which means no characters have been received by the serial I/O device. When a character has been received by the serial I/O unit, this status bit goes high, returning a one and allowing control to pass to the next statement.

Statement 460 reads a byte from the data register of the Z80-SIO chip. The AND 127 statement masks or turns off the top bit of the byte just read, yielding a single 7-bit ASCII character into X$. This character will ultimately be displayed on the CRT of the local microcomputer, letting the user know what is being transmitted.

We don't have to do anything special to send data out the printer port to the modem; we just use the LPRINT statement that Microsoft BASIC provides. For reading the echo back from the printer port from the modem, a more complicated method is required. As you can imagine, very few systems would anticipate the desire to read data back from a printer.

No matter, most systems use a serial I/O device that has both a transmit and a receive section, and both sections are probably connected to the appropriate serial printer port. For our communication purposes, the serial printer port is connected to a modem and the modem to the telephone line. Discussion of the modem connection is beyond the scope of this article.

The BigBoard II microcomputer uses the Z80-SIO serial I/O chip for all serial I/O, both to/from the console terminal and to the printer. This discussion generally applies to other serial I/O chips or USARTs in use (i.e., the i8251, ns8250, etc.). In this particular configuration, I/O read address 80 (hex) is the I/O read address for channel A or the printer (LST:) port, with 82 (hex) being the I/O read address for channel B or the terminal (CON:) port. This is summarized in Figure 1 (page 64). The bits in the status register are described in Figure 2 (page 64).

What interests us here is bit D0, which is a one when a character has been received, indicating we are ready to read from the receive data register. So we loop on this bit until it turns from a zero into a one, at which time we read the character just received at the receive data register of the SIO.

As an example of how these programs might be changed to conform to other serial I/O devices, the i8251 has its Receive Character Available bit in D1, not D0. For this chip, you would AND the status byte read with two in order to set all bits read from the status register of the 8251 to zero (except bit D1). The IF statement here should test for <>2, not <>1.

There are several possibilities for enhancing the ideas expressed here. Error checking could be performed by comparing the return echo character with the original character. If they are different, there was an error in transmission or reception. If an error is detected, send a backspace and resend the original character. Of course, you won't know whether the error was in the link to the host or from the host to your microcomputer. To speed up transmission, try sending a space and then a backspace before each line so that transmission and echo can overlap. You'll also find that the techniques in this article will assist the direct transfer of data from one microcomputer to another, without a telephone modem hookup. Hopefully, these remarks and programs will be helpful to you as you telecommunicate with your microcomputer. DDJ

100-130	Initialize the environment.
140-170	Enter dialog with user to gain the filename to transfer and open it as an input file.
200-360	The main body loop of the program.
370-400	Report statistics of the file transfer to the user and determine whether another file transfer is wanted.
450-470	A subroutine that checks the status register of the serial I/O device and reads a byte of data from the data register when a character has been received.

Table I
Statements

(Listing begins on next page)

BASIC Telecommunications Programming (Text begins on page 64)

Listing One

```
100 REM "ToUnix" transfers data files to unix cpu running cat, from cp/m
110 REM                                  printed_________@________
120 WIDTH LPRINT 255          'no system carriage returns
130 TRUE=-1
-------------------------------------------------------------

140 PRINT"ToUnix here"
150 PRINT"Enter file name to transfer ";
160 INPUT FL$
170 OPEN "I",1,FL$            'open the input file

-------------------------------------------------------------

180 CHARS=0
190 LINES=0

-------------------------------------------------------------

200 IF EOF(1) THEN 370        'check for end of input file
210 LINE INPUT #1,A$          'read next line from input file
220 IF LEN(A$)<1 THEN A$=" " 'if a null string send a space anyway
230 LINES=LINES+1             'count statistics
240 A=LEN(A$)
250 CHARS=CHARS+A             'count statistics
260 FOR I=1 TO A              'for all the characters in the line
270   LPRINT MID$(A$,I,1);
280   GOSUB 450               'wait for echo back from unix computer
290   PRINT CHR$(X);          'display it on the terminal
300 NEXT I
310 LPRINT CHR$(13);          'send the carriage return to end the line
320 GOSUB 450                 'wait for character received from modem
330 IF X<>10 THEN 320         'line feed received, the prompt for next line
340 LINES=LINES+1             'one more line sent
350 PRINT
360 GOTO 200                  'get next line from input file

-------------------------------------------------------------

370 CLOSE
380 PRINT                     'print statistics of the transfer:
390 PRINT "File transferred"
400 PRINT CHARS;" Characters transferred"
410 PRINT LINES;" Lines transferred"
420 PRINT CHR$(7);            'beep to operator that transfer is completed
430 GOTO 150                  'get next file to transfer, if any
440 END

-------------------------------------------------------------

450 IF((INP(129) AND 1) <>1) THEN 450 'wait for data to read back
460 X=INP(128) AND 127        'readback the echo from the host computer port
470 RETURN
```

End Listing One

(Continued on next page)

BASIC Telecommunications Programming (Text begins on page 64)

Listing Two

```
100 REM "ToHP" Transfers text files to H-P computer running edit2, from cp/m
110 REM                                          PRINTED
120 WIDTH LPRINT 255          'no system carriage returns
130 TRUE=-1

------------------------------------------------------------

140 PRINT"ToHP HERE"
150 PRINT"ENTER FILE NAME TO TRANSFER ";
160 INPUT FL$
170 OPEN "I",1,FL$
180 CHARS=0
190 IF EOF(1) THEN 350        'check for end of input file
200 LINE INPUT #1,A$          'read a line of data from input file
210 IF LEN(A$)<1 THEN A$=" "  'if a null string then send a space anyway
220 LINES=LINES+1
230 A=LEN(A$)                 'how many characters are in this line
240 CHARS=CHARS+A

------------------------------------------------------------

245 LPRINT CHR$(17);          'send leading control Q...
250 FOR I=1 TO A              'for all the characters in the line
260   LPRINT MID$(A$,I,1);    'send the next character in the line
270   GOSUB 430               'wait for echo back from the H-P computer
280   PRINT CHR$(X);          'display the echo on the local terminal
290 NEXT I
300 LPRINT CHR$(13);          'send the carriage return to end this line

------------------------------------------------------------

310 GOSUB 430                 'wait for character to be returned from modem
320 IF X<>17 THEN 310         'discard characters until the control Q prompt
------------------------------------------------------------

330 PRINT                     'local carriage return
340 GOTO 190                  'get the next line, if any

------------------------------------------------------------

350 CLOSE                     'all done with this file,
360 PRINT                     'report status and statistics to user
370 PRINT"FILE TRANSFERRED"
380 PRINT CHARS;" CHARACTERS TRANSFERRED"
390 PRINT LINES;" LINES TRANSFERRED"
400 PRINT CHR$(7);
410 GOTO 150                  'try to get another file to transfer
420 END

------------------------------------------------------------

430 IF((INP(129) AND 1) <>1) THEN 430 'wait for data to echo back
440 X=INP(128) AND 127        'readback the echo from the H-P computer port
450 RETURN
```

End Listing

U.S. Robotics' S-100 Modem

We found this 1200-baud S-100 plug-in modem very interesting – it's the only one we know of. We reserved detailed examination of the product for an upcoming review, deciding instead to let someone intimately associated with the product give you a look at the process of its creation. – Ed.

About ten years ago I used my first modem. It was a large gray box in a drawer of a special desk holding an IBM Selectric-type terminal. This modem communicated with a large mainframe computer at the lightning speed of eleven characters per second. Since this replaced punched cards, the slow speed did not seem so bothersome. But one day a thirty-character-per-second modem was installed: this was high speed computing.

Ten years later, I am sitting at my own computer with a modem installed in the machine and I can communicate with other computers over the telephone network at over 120 characters per second. I never thought I would own a computer, let alone work for a company that manufactures modems.

The last ten years (for data communications, the last two or three) have seen tremendous advances in telecommunications software and hardware. Today, both the computer and the modem are priced within the reach of the average computer user.

Less than a year ago, U. S. Robotics decided to redesign the Autodial 212A unit which was the top of the line product produced by the company. The main goals of that effort were to produce a smaller, less expensive product and a simpler-to-use modem. This product would help to make 1200-baud intelligent modems more available to the average computer user.

Most of the auto dial units then on the market looked like many of the early home computers. They included many lights and switches. The average modem user, like today's average computer user, usually needs an on/off indicator and an on switch to turn the unit on and off. The lights are useful for real technicians, but most people could care less.

In addition to the simpler user interface, the design team developed the concept of a core modem. The core modem circuitry contained all the electrical modem elements for accepting TTL-level digital signals and producing analog signals necessary for telephone transmission. By adding a power supply, RS232 interface, and other circuitry, this product could be customized for other equipment manufacturers or made into specialized products.

Several of the members of the engineering team were long-time users of S-100 systems. After the production of the Password, the new redesigned modem, the attention of several people turned to other products to incorporate the core modem concept. Among the first was the S-100 modem. Although there were several S-100 modems available, there were no intelligent 1200-baud modems for S-100 computers.

Advantages and Disadvantages of the S-100 Modem

The S-100 card modem presents several advantages for the user. For people like me there is the question of space. A card modem is integral to the computer and does not take up any of the work area around the computer.

Cables, it seems to me, are the Gordian knot of computer owners. For most modems, at least three cables are needed: power, telephone line, and RS232 connector. The power for the modem comes from the S-100 bus, so there is no adaptor to plug into an already overcrowded power strip. Communication between the modem and the computer is over the S-100 bus which eliminates the RS232 cable. The S-100 modem requires only the telephone cable.

The elimination of the RS232 interface also removes the problem of cabling between the modem and computer. Removing the necessity for an external RS232 cable also eliminates the need to use one of the serial ports on the computer for communications.

Today many of the modems on the market are intelligent modems. These modems are much simpler to use than many of the earlier models. For the most part they are controlled by commands sent over the serial communications line. Commands may be incorporated to dial numbers, turn on and off speakers, send result codes, and numerous other functions. The commands provide a means for software control of all the modems' functions. In terms of control, the S-100 modem is the ultimate intelligent modem. There is no on/off switch; it is a fully integrated part of the computer.

Systems developers or integrators can develop programs which can establish a communications link without the user of the system ever knowing the telephone number dialed or log-on sequences required. This is possible through the use of the autodial function and direct input/output over the S-100 bus. Sophisticated users may be able to circumvent such techniques.

There are some disadvantages to the S-100 modem. Since it is not a separate unit, it cannot be easily moved from one machine to another like a stand-alone unit. It may be slightly more complicated to install the S-100 modem for the simpler types of communications. Most systems provide some driver routines for serial communications devices, but the S-100 may require user-written driver routines.

The decision to use an external modem or an integrated modem depends on the needs and preferences of the user more than on the units themselves.

The S-100 Modem Design Process

The design team started with the core modem, which they now refer to as "essence of modem," and began to design a card modem for S-100 systems. One of the first factors taken into consideration was the S-100 standards. The design team had had experience with S-100 cards in the past and knew the troubles with non-standard cards. The first requirement of the new design was that it be compatible with the new IEEE 696 standard for S-100 systems.

The bulk of the design work then centered on the serial interface. Several important criteria came into play when designing the interface. First was the availability of the parts to be used (were several sources available?); familiarity with serial interfaces (what experience did the designer have with the parts?); and simplicity of operation (would the user be able to operate the modem with a basic knowledge of S-100 systems?).

The 8251 USART (Universal Synchronous/Asynchronous Receiver/Transmitter) was chosen for the basic design because of availability of the part and the

by Michael McKillip

Mike McKillip, U.S. Robotics, 477 E. Butterfield Rd., Lombard, IL 60148.

design team's familiarity with it. The S-100 card modem would operate as a fairly independent unit so that all timing and interface circuitry would be contained on the board.

Since most S-100 systems are based on the 8080 or Z80 family of microprocessors, the input and output for the modem were ported rather than memory mapped. The current design provides for 32 possible locations in the port addresses. S-100 standards provide for optional extended addressing for 16-bit port addressing, but this was not incorporated into the current design since 16-bit processors are currently more the exception than the rule. This will probably be changed as 16-bit processors become more common.

After the production of the basic specifications of the product, the first prototype was produced. This prototype incorporated the core modem and a wire-wrapped board containing the circuitry for the serial interface, speaker, and telephone lines.

To test a prototype and debug a wire-wrapped board is good for the soul. If you haven't done it, you're missing a lot of fun. If you have, you know what goes on. The first step in the process is to turn off the computer and take out every card on the bus. Usually an extender card is placed in a slot and the wire-wrapped board placed in another slot. The prototype card is then placed in the extender card. Carefully, you turn the computer on and keep one hand on the switch. As soon as the power is turned on, you look for smoke. If the prototype is "smoking," you quickly turn off the computer. This may sound funny, but ask anyone who has tested prototypes and they can tell you about at least one new board that smoked.

No smoke is the best thing the designer can hope for the first time a new board is powered up. If there is no smoke, the board should sit in the system for about one hour or so with the power on. This provides a chance for any component which may have a problem to burn in a bit. After about an hour, each component is checked for heat. Usually a finger on the part is enough to tell if there is a problem. Then a volt meter and an oscilloscope are applied to various circuits to check the voltage levels and pulses. Wire wrapping can result in bad connections that are tough to find.

The S-100 modem only had two wire-wrapping problems the first time it was tried. The real problem with wire wrapping is the transient problem: sometimes the board will work and sometimes it won't. Trying to track that problem can be a hair-pulling experience.

After the smoke test, the system is powered down and the other cards reinserted. The system is turned back on and the new card tried for the first time. With the S-100 card, it worked the first time. Data sent to the serial interface went to the modem and it worked. Several days of testing did show some minor flaws in the wiring, but they were easily corrected.

The next stage in the development of a new board is to have real circuit boards made. This requires developing a mask for photo etching and having cards actually printed. The S-100 card modem is a fairly simple design which requires only two layers (front and back of the card). The actual layout of the circuitry by a layout artist required about a week, and then about another week was required for the cards to be produced. The first real test of the product was in the printed circuit card.

After the cards are returned, a visual inspection is made, the components are soldered into place, and the test begins again. The smoke test is repeated. If the modem passes this test, then the full testing begins. Again, the S-100 modem required only minor changes. Two of the printed circuit trails were too close to each other, and one trail was on the wrong side of the board.

Correcting the problems was a simple task, but to insure a quality product one more round of printed circuit prototypes was developed and tested. These passed the test and production of the product began.

The chief designer of the S-100 modem is a good engineer who hates to write. So the hardest part of the development for him was the manual. Each designer is required to write the preliminary version of the manual which then goes through several revisions by the technical writing staff and the customer service staff before a version is sent to customers. After about six months, the version provided to customers is thoroughly reviewed and revised based on customer service feedback.

Design of the S-100 Card Modem

The photograph in Figure 1 (page 74) is a picture of the U.S. Robotics S-100 card modem. The S-100 modem may be divided into two functional parts: the modem and the serial interface. The modem is the functional equivalent of the U.S. Robotics Autodial 212A or Password modems. It is a fully functional autodial/autoanswer modem which operates at 0 to 300 baud and 1200 baud, according to the 212A protocol. (See below for more information about baud rates.) Instead of the RS232 connection between the computer and the modem, the S-100 communicates with the computer through the S-100 bus. Communication with the modem takes place through the commonly used I/O channels of the 8080, Z80, and 8086 family of microprocessors. Serial communication is through a 8251 USART.

The Modem

The S-100 card modem is an intelligent modem with autodialing and autoanswering capabilities. The S-100 modem accepts commands over the serial interface in the form of ASCII text characters, numbers, and special characters. An example of a command is:

ATDT 555-8989

This command would autodial the telephone number 555-8989. Commands may be given to the modem for the following functions:

Enter Answer Mode
Dial
Echo back characters
Half/Full Duplex
On/Off Speaker
On/Off Result code
Set Number of Rings Before Answer
Change Disconnect Command Characters
Set Wait for Carrier time
Set Terse/Verbose Response
Use Extended Result codes
Reset

Once a communications link has been established, all information sent to the modem is passed over the communication lines. The only command the modem will accept is the command to disconnect (+++). The +++ sequence must be preceded and followed by one second of no data transmission. The character used as a three-letter sequence for disconnect can be changed by the user.

The S-100 modem will also automatically switch between originate and answer modes. When answering the telephone, it will automatically select between high-speed communication and low speed. Codes are sent to the terminal device indicating whether a high-speed or low-speed connection was established. The microcomputer can then adjust its own baud rate accordingly.

The S-100 modem also contains a speaker (upper left in the photograph in Figure 1) which will indicate the sound of the dialing tones and provide information about the status of the connection as it progresses. Once communication has been established, the speaker will automatically turn off. The user has the option to turn off the speaker by issuing a command. Although the speaker is contained within the case of the computer on the S-100 modem, it is generally loud enough to be heard through the computer case.

Baud Rates

The 212A communications protocol for modems provides for the transmission of data at 0 to 300 and 1200 baud. 600 baud may be found among Remote CP/M systems and other bulletin boards. The

S-100 card modem has been designed so that it can handle 600-baud communications.

Since the serial interface and modem are contained on one card with no external RS232 interface, some of the signals designed for use with RS232 may be used for other purposes. In this way, the Request to Send line, sometimes used as an RS232 signal, has been used in the S-100 modem to provide access to more baud rates. When programming the 8251, if the RTS line is set to low, the baud rate will be divided in half. In this way, 1200 baud may be switched to 600 baud and 300 baud to 150 baud.

It must be remembered that 600 baud is really a fast 300 baud and not a slow 1200 baud. When a telephone connection is made, the protocol's baud rate is determined by the sounds the two modems make. These sounds follow a set pattern known as a protocol. The 300-baud and 1200-baud protocols are very different. Once a connection has been established at 300 baud, it is possible to change the baud rate to 600 baud at both ends and continue communications. Both sides of the connection must operate at the same baud rate or else garbage information will be received.

The design team believed that the large number of RCPMs and the common set-ups for communications at 600 baud required the inclusion of 600-baud capabilities.

The design process for the S-100 modem from beginning to the time the first products were shipped required about ten weeks. One of the real joys of helping to develop a new product is to see it boxed up and shipped out the door. There are other advantages. The chief designer of the team has serial number 1o of the modem hanging on the wall of his office and takes pride in showing it to visitors. Serial number two of the modem is in the computer used to write this article and was used to send the copy to *Dr. Dobb's* for printing. **DDJ**

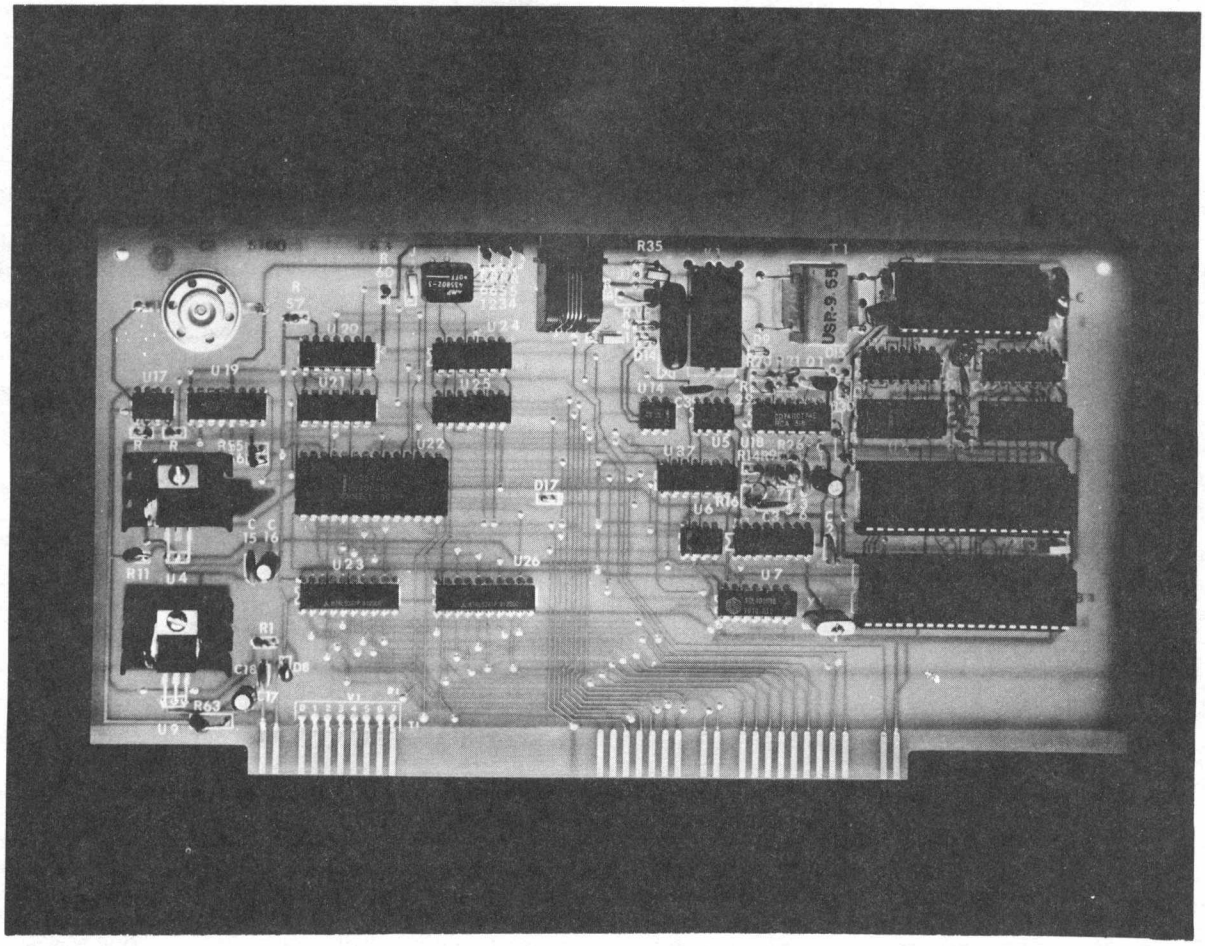


Figure 1.

Friday!

Company: Ashton-Tate, 10150 W. Jefferson Blvd., Culver City, CA 90230
Computer: CP/M 2.x, CP/M-86, and MS-DOS
Price: $295.00

Reviewed by R. A. Langevin

Friday! by Ashton-Tate is the son or, if you prefer, the daughter of dBase II. It provides most of the features of dBase II without requiring any knowledge of programming on the part of the user. In some respects Friday! provides capabilities that go beyond those of dBase II, since it permits users to custom design multi-line reports, create form letters with inserts from a data file, and define and print mailing labels – features not available in dBase II.

Environments

Friday! is available for a variety of current processor and operating system environments. These include 8080, 8085, and Z80 processors with CP/M 2.0 or higher and 8086 and 8088 processors with MS-DOS or CP/M-86. Memory requirements depend on the operating system used. A minimum of 64K is required for CP/M-80 and 128K for MS-DOS and CP/M-86.

In a floppy disk environment, dual floppies are required with a minimum of 126K available storage space. Friday! can also be used in a hard disk environment. A terminal or monitor that supports at least 80 columns and direct cursor addressing is required. Any printer can be used as long as it supports line lengths of 80 characters or more.

Features

Friday! is designed to support record-oriented files. A record can consist of up to 32 fields and can be up to 999 characters long. There are no provisions for files that span two or more disks, so the maximum file size that can be accommodated is limited by the space available on a single data disk.

All interactions with Friday! are by means of an extensive set of nested menus: there is no integral programming language as with dBase II. As a result no programming knowledge or experience is required to use any of Friday!'s facilities. The menus support the design of a file; entry, updating, deletion, sorting, selection, viewing, and printing of records; creation and printing of standard and custom reports; preparation and printing of form letters that include data from a selected file; and specification and printing of mailing labels.

In addition to these impressive capabilities, Friday! data files are compatible with dBase II and Lotus 1-2-3 files and can be read or written to by WordStar. Since the format of Friday! files is well defined, users can access them using the facilities of BASIC, Pascal, C, and other programming languages, a capability that will not normally be required.

After a file is designed, a standard Friday! data entry screen can be used or a custom-designed data entry screen can be readily produced with a series of screen layout menus. This is a useful capability. The user not only can design screens that are esthetically more pleasing than the standard data entry screen, but more importantly can create a number of custom screens for a single file. Each screen can be designed to present only the fields desired, so that selected information in the file (for example, salary information) is hidden from the individual doing file updates.

Two extremely useful features are available when creating or updating files. The first, a "ditto" capability, premits the user to specify a field or fields that contain fixed information (for example, a date or a price). This fixed information is automatically carried forward to each record as it is entered or updated, greatly speeding up the process. The second feature permits any field except the first to be a calculated field. Formulas for calculation of individual fields can be up to 60 characters long and have available a maximum of 500 characters. More important, the values of calculated fields are saved in the data file being worked on!

Reports can be created in two forms and the formats saved on disk. The first and simplest, the Quick Report, enables the user to specify the fields to be listed; give them titles that may differ from their internal names; specify their sequence across a report line; and print the corresponding report using up to 15 relational rules to select the records to be used. Although Friday! always maintains the records in a file sorted on the first field of the record, temporary sorts can be defined on as many as five fields for the production of reports.

The second and more flexible report form, the Custom Report, permits the user to select the fields to be printed and to locate them anywhere on the page. This flexibility, not available in dBase II, affords the opportunity for considerable creativity in report layout. Since the Custom Report format can incorporate any text whatever in addition to data from the records selected, it can also be used to create simple, single-page form letters, as well as be adapted to the creation and printing of mailing labels.

Documentation

Friday!'s documentation is among the best I've ever seen. Its 207 pages are attractively printed and bound in a sturdy three-ring notebook. The back of the notebook is hinged so that it will stand up on a desk. All in all, very attractive packaging!

The documentation consists of a brief overview of Friday! and a series of six lessons that should be worked through at a terminal. These lessons will quickly teach anyone how to use all of Friday!'s features. The next section of the manual, almost 75 pages long, provides an exhaustive description of all the prompts that appear on the screen as Friday! is used. A glossary is also provided to help new users over the rough spots of terminology.

A series of eight appendices and a good index conclude the documentation. The appendices describe Friday!'s sorting order, file naming conventions, relationship to dBase II, field renaming procedures, use of backup files, working and backup copies of Friday!, installation instructions, and the use of Friday! with Lotus 1-2-3. After the index a handy reference folder summarizes Friday! commands and procedures. This is a documentation model that others could well follow.

Problems

Unfortunately, nothing is perfect. There are a few things in Friday! that I would like to see changed.

Although the user can create a custom screen for data entry or update and can place fields anywhere on the screen, the sequence that the cursor follows as it skips through fields is the sequence in which the fields were originally defined in the record layout. While this is not a real problem, it can be disconcerting to see the cursor jumping around the screen.

I consider the two built-in, unchange-

able default limits to be a more serious problem. Specifically, it is impossible to print a line longer than 160 characters, and the physical length of a page is always taken to be 66 lines. Many printers are capable of printing lines substantially longer than 160 characters, and the extra width can be extremely useful. It's not clear why Ashton-Tate chose such a short line length as an upper limit. The built-in assumption of 66 lines per page can be lived with, but many times it is more convenient to print a report on paper that is 8½ inches high and 11 inches wide; this is difficult to do with Friday!.

Summary

The foregoing criticisms aside, Friday! is a powerful file handling package that affords most of the capabilities that a user would normally want. Best of all, these capabilities are now accessible to the nonprogrammer. Friday! should do well in the marketplace.

SAL/80

Company: Protools, 24225 Summerhill Avenue, Los Altos, CA 94022
Computer System: CP/M 2.2
Price: $59.95

Reviewed by Jim Kronman

SAL/80 is a software package that adds standard "structured" control operators to the 8080/85 assembly language and can also, at the user's option, produce some code optimized with Z80 instructions. SAL/80 is designed to be used with the Digital Research, Inc. MAC and RMAC macroassemblers. In addition to the control structures, built-in library functions are provided. Newberry Microsystems refers to SAL/80 as a "language," but it is actually a language enhancement, used in the same way as RATFOR is used with Fortran.

Structured programming is a lot like religion: either you believe or you don't. I will not try to sell you on the concept in this review, but if you believe in minimal use of GOTO, you will appreciate SAL/80.

This review applies to version 2.1 of SAL/80 running under CP/M 2.2. The author of SAL/80 is Steve Newberry and it is marketed by Steve's company, Protools. The current price for SAL/80 including a 200-plus page manual is $59.95. The price includes a free update to version 3.0. SAL/86 is or will be available for 16-bit processors.

What It Is

SAL/80 provides these familiar control structures:

```
DO/ENDDO
FOR/UNTIL/STEP/ENDFOR
IF/ENDIF/ELSE/ENDELSE
LOOP/UNTIL/ENDLOOP
REPEAT/ENDREPEAT
WHILE/ENDWHILE
SELECT/CASE . . .
    CASE/ENDSELECT
```

EXIT and GOTO are also provided.

The control structures are supported by a wide range of boolean expressions of the form X,rel,Y where X and Y (with a few limitations) can be any register, register pair, memory reference, or constant. The "rel" in these expressions can be either the usual comparisons (such as greater than, equal to, etc.) or one of several CPU status flag conditions (e.g., carry set, minus, etc.) that require no argument (X and Y are null and the PSW is implied). For convenience, IFCALL and IFGOTO are provided to allow a conditional call and goto consistent with the other operations.

The SAL/80 built-in functions include register save and restore functions. Most of the built-ins are designed to facilitate program-to-user interface via the console.

Several SAL/80 library files are also provided on the distribution disk. They include block move, string search and comparison procedures, and multiply and divide.

SAL/80 is designed to work in a 8080/85 environment but optionally will generate Z80 code for relative jumps and the "DJNZ" instruction in loops.

Using SAL/80

SAL/80 version 2.1 is implemented as a set of macros that must be invoked as "MACLIB SAL/80" at the beginning of your assembly language program. You include the SAL/80 instructions as part of your assembly language program code. When it is time to assemble the code, SAL80.LIB and any of the provided ".LIB" files you have referenced must be on the same disk as your source code. You run MAC or RMAC and the program is assembled.

SAL/80 Documentation

The SAL/80 manual has four chapters and three appendices. The chapters are indexed. The compiler source code has its own index, this source listing plus its index taking 72 pages.

Chapter One, "Overview of Essentials," explains the syntax, content, and use of SAL/80 in 45 pages. A generalized example of each construct or function is given in schematic form. For example:

```
LOOP?
    [ < loop body > ]
UNTIL? < bool expr >
    [ < more loop body > ]
ENDLOOP?
```

This example points out one feature I have so far not mentioned: all SAL/80 instructions are terminated with "?".

Chapter Two, "Tutorial: Maintainable Programs," is a 31-page essay on software design using a real-life (if somewhat contrived) example. All of the steps from design concept to finished product are covered.

Chapter Three, "Tutorial Worked Example: MEMTEST," is the complete source code listing of the example used in the previous chapter. It includes its own index (for the program) and occupies 50 pages. MEMTEST is a memory test program.

Chapter Four is called "General Observations" and talks about using MAC and RMAC and the application of some of the features of SAL/80.

Appendix A of the manual is the complete SAL/80 source code. The distribution disk file SAL/80.LIB, however, does not look like the source code. It has been, in Newberry's words, "krunched" in order to save disk and TPA space. Appendix B lists the source code for the error trapping portion of SAL/80.LIB. Appendix C is a summary of SAL/80 commands.

Performance

SAL/80 does what it claims to do. It runs, has a few bugs, but is well supported by the author. I can only fault it for being *slow*. On the other hand, RATFOR is no burner with Fortran, either. Steve Newberry has recognized the speed problem with his MAC.LIB approach and is currently preparing a preprocessor version of SAL/80 written in the C language. Purchasers of version 2.1 will receive version 3.0 free when it is available.

The code produced by SAL/80 is reasonably close to being as compact as the code you would produce if you coded in "pure" assembler. For example, a small menu-driven program I wrote with SAL/80 to send set-up commands to a printer was 1.4K bytes, and a similar program for another printer from "pure" assembly code was just under 1.0K bytes. Although some minor inefficiencies have been created in exchange for generality, SAL/80 generates more compact code than any level language compiler I have used. I have not noticed any case where execution speed has suffered. My experience with SAL/80 has been with business applications; if you are coding a real-time control application, however, you might encounter critical areas where fine tuning is required. The same would be true if you were coding in "pure" assembler.

Most of the constructs work as you would intuitively expect them to from experience with higher level structured languages such as Pascal or C. I found the

SELECT construct, however, more limited than its CASE counterpart in Pascal. SELECT/CASE. . .CASE/ENDSELECT is actually more like Fortran's computed GOTO statement. The first choice (CASE) *must* be 0, then 1, and so on. This is stated in the documentation, but I had a hard time understanding what it meant, probably because I wanted it to work like Pascal! Another enhancement that would have been nice is if the X,rel,Y operations (relational operators, *rel*) would accept the syntax "A" to represent 41 hex and so on (like CHR(A) in Pascal or the literals in standard Intel assembler code).

I have not succeeded in getting the example program, MEMTEST, to run on my computer. I must admit that I have not given much priority to this and therefore have not spent much time trying. I have had no substantial difficulty, however, getting any of my own applications written with SAL/80 running.

The only other operational criticism I can make is that the error messages are not always sufficiently clear to allow debugging without resorting to examination of a ".PRN" file with the macro expansions included. This might not seem like a big deal until the first time you look at such a file from SAL/80. Some of those innocent-looking constructs and built-ins can generate gobs of code!

I have found no problems in the built-in and library functions supplied with SAL/80, but in some respects they are a reinvention of the wheel. My personal preference is for a set of routines like Richard Conn's SYSLIB functions, which are in the public domain. I find some of the SAL/80 procedure names (such as "compare$strings:") to be longer than necessary; I type a lot, but that doesn't mean I like to do it.

Evaluation of the Documentation

When I originally ordered SAL/80, I was motivated to do so by the documentation more than by the program itself. I thought it would be an economical way to learn more about MAC and RMAC and to pick up a few new programming tricks. My aims were accomplished, but I feel the documentation could be improved in some areas.

The manual is better than many. It *is* thoroughly indexed. In fact, I wish more software publishers would be as thorough as Steve Newberry has been in indexing his manual.

The greatest improvement in the manual would be to provide a real, concrete, illustrative example with the definition of each command. This would go a long way toward making SAL/80 immediately useful. I found I had to try each construct a few ways until I got clear in my mind what each function could and could not do. Some real examples accompanying each function description would have saved me hours of time.

Another welcome addition would be a simple, columnar quick-reference list of all reserved words and a compact quick-reference guide for the commands. I would like to see the commands, built-ins, and so on defined one to a page in the manual for ease of reference.

Support

I have nothing but praise for Newberry's support of his product. I am somewhat reluctant to print constructive criticism about SAL/80 because he has been so responsive to past inputs (from me and others) I am reasonably sure that most of my criticisms will have been answered by changes in the documentation by the time you read this review. His initial release of version 2.1 had some bugs in it, but it was released at a discount. In addition, anyone who reports a real bug receives a free update to version 3.1 for the effort (version 3.0 is already included when you buy 2.1). Corrections fixing the identified bugs have been mailed to users in a timely manner.

Conclusions

If you must do assembly language programming and want to generate more readable and maintainable code, then you should investigate Protools' SAL/80. It has the potential to give you some of the advantages of the higher level languages while retaining the control and efficiency of machine-level programming.

FILEBASE

Company: EWDP Software, P. O. Box 40283, Indianapolis, IN 40283
Computer: 64K Z80 CP/M
Price: $75.00

Reviewed by Dennis Cashton

The object of many user-written programs, as well as many purchased software packages, is the manipulation of data in one form or another. Quite often what happens to data is not exactly what you had in mind, and you wind up having to write some sort of fix-up program to massage the data. Or you have a data entry package, a prime example of one application that cannot stand alone, and want to write a program to handle the data after entering them. On the way to your accounting programs or customer file processing programs, you are bound to run into a need to sort, extract, globally change, or otherwise shift things around in a file. In all these cases, it would be nice to have a listing of the data as they came from your source file or after they are sorted. Well, it seems that the folks at EWDP Software have a pretty good answer to your problems.

FILEBASE is an effective utility for performing many useful file handling functions. With its clean, no-frills, menu approach, the user can select to:

1. SORT a file (no exclude or include)
2. SELECT/EXCLUDE only (no sort)
3. SORT with SELECT/EXCLUDE
4. LIST record positions and fields (to screen or printer)
5. ADD new fields to an existing file
6. CREATE a new file
7. APPEND data to an existing file
8. WRITE a subset of fields to a new file

In going through the manual for FILEBASE, the first thing I noticed was that the people at EWDP have gone to a lot of trouble to make the documentation and the program very easy to use. They assume no prior knowledge on the part of the user, and you are led by the nose through every function. Examples are provided for each function described, and explanations for every example.

It would be possible to load the program and use it without reading the documentation, but some of the features that make FILEBASE such a versatile tool would not be obvious from the menus provided. One example of such a feature is the ability to sort on a last name within a name field by using the "L" suffix on a field number. You have the same capability to get at zip codes by specifying a "Z" suffix.

It's little extras like this that make FILEBASE nice to use and especially suited for mailing list applications. In fact, the nice people at EWDP Software sent a draft of a brochure describing FILEBASE in such an application. Especially nice is the feature that allows you to set a pause between each record on printing, as well as the number of lines to skip between records and the tab position for start of printing. This makes it very simple to print envelopes. Other print controls make it easy to handle any size or shape of label, card, envelope, or paper.

Other niceties include the ability to merge input files and to create multiple output files from one or two inputs. Comparators for SELECT/EXCLUDE can be:

EQ - equal to
GT - greater than
GE - greater than or equal to
LT - less than
LE - less than or equal to

(now the good stuff)

BT - between values
LS - one of a list of values

RR - range of record numbers

Sort can be ascending or descending, and the number of fields that can be printed across the page depends on the specified printer paper width.

There are, as always, several cautions and catches to watch out for. First, the documentation I received was stamped "OUTDATED MANUAL – SAMPLE ONLY" and was for version 1.1. The diskette supplied was labeled Release 2, and an attached note stated that the Release 2 manual would follow on publication. I can only hope the functional differences are not significant.

The manual refers often to the use of "comma delimited fields," but this is standard for many data entry programs and most versions of BASIC. The use of files "external" to FILEBASE is also stressed. All this means is that your input files will not be manipulated or destroyed by inadvertent bungling. In my many years of experience with inadvertent bungling, I consider this a definite plus.

Unlike the common data base management systems, FILEBASE does not need to have fixed-field, structured records. It will actually read the file to find out the field types and lengths.

FILEBASE has its limitations, but it is still more than sufficient for most applications. The limits are:

Record length - 4096 bytes max
Fields per record - 40 max
Input files per execution - two
Output files per execution - two
Records for SELECT/EXCLUDE - 15000 max
Records for sorting - 4000 max
File size - 240K max (more possible, but requires extra processing time)
Numeric sort precision - 10 digits

One interesting note: The manual says it needs 64K to execute, but it ran with no noticeable degradation on my 52K CP/M system.

I find it difficult to express how pleased I was from the moment I put the disk in and typed in FILEBASE to get it started. Everything worked as stated, if not better, and I instantly thought of a dozen things it would come in handy for. When you weigh utility against price, I don't see how a person who deals with any kind of data files could do without this program. It is everything you could ask for, in documentation, function, and price.

EWDP has informed us that Release 3 will be introduced at SOFTCON in New Orleans, Feb. 21-23. Enhancements include random access by user controlled indexing of any field and ability to select or exclude records by sub string searches. – Ed.

Oubliette

Company: Centaur Software, 501 Jackson, Charleston, IL 61920
Computer System: CP/M
Price: $39.95

Reviewed by W. James Wiggins

Venturing down a dark passageway, you come to a doorway. Peering cautiously in, you see before you a large bronze dragon; he suddenly spies you and . . . the battle is on.

This could easily be the scene in the adventure game called Oubliette. According to Centaur Software, Oubliette is French for dungeon or maze. The game follows the type of game made popular by TSR, the Dungeons and Dragons series – and follows it very well.

This author has been addicted to adventure games for about two and a half years now and has played "several," those with graphics and those without. When Oubliette was given to me to review, I therefore felt confident that the game would be a snap to play, having had so much experience before with "games of this sort." I therefore sat down to play it without any more than a cursory stare at the artwork on the front of the very nice looking manual (after all, who reads manuals or instructions?). After wiping out my entire party (fortunately on my backup disk) I decided it might be good to at least look up some of the magic spells. Two hours later, I finally got back to the game . . . not because the manual was that hard to understand – it was that engrossing.

Centaur documents not only why the world of Oubliette is the way it is, but they even translate the magical spells so that we lesser mortals can understand them. The manual flows well – there are charts and graphs clearly and succinctly comparing various races and classes; there are in-depth explanations on magical and clerical spells; there is even an almost complete map of the first level of the dungeon.

The monsters, races, and classes will seem familiar if you have played Dungeons and Dragons before; the thrill of being able to run up to six characters by yourself and have the Dungeon Master be totally impartial will hook you on this game, though.

The only drawback to the manual is the documentation of how to set the game up for your terminal. The disk includes a program to help you configure Oubliette for your terminal, and unless you have a strange terminal like I do (I was playing it on an Apple II with a Videx 80-column card), you should be able to set it up for your terminal very easily.

Now that I am an expert at the game, let me tell you the best way to play it. First, you need to build up your party and then . . . well, maybe I'll let you figure it out (maybe I will too). The one thing that I will tell you is that when I called and spoke to the people at Centaur, they asked how far down in the dungeon I had traveled. I admitted to going down to only the third level. They said that most people who went down farther seldom lived to tell about it. I wonder what's down there?

In case I haven't given you a clue, the game is very good – realistic, engrossing, and well thought out. I would say that it is the most engrossing game that I have played in a long time. ■■J

(Continued from page 28)

performance.

If you need to interact with an IBM mainframe, but don't need to move much data, look into 3270.

As with everything else, there are good packages available and also some very poor ones; investigate before you buy.

I have waved my hands over some of the details and completely ignored others, but I hope that overall this has given you a pretty good introduction to communications. If you now know the proper questions to ask and can tell when someone is trying to snow you (or just doesn't know the answer) then I will have accomplished everything I could reasonably hope for. I have no relationship to any of the companies or products mentioned, except for HASTE which I helped develop. ■■J

BOOK REVIEWS

C Programming Guide

by Dr. Jack Purdum
Published by Que Corporation
$17.95, 250 pages
Reviewed by Dr. Joseph B. Rothstein

Just a few years ago, programming a microcomputer in a high-level language meant one thing: BASIC. Now that programmers can take advantage of microcomputer implementations of Pascal, Fortran, Lisp, APL, Forth, Ada, Cobol, and others too many to name, it appears likely that no language will again have the dominant position BASIC has enjoyed.

But from this Babel of programming languages, a recent entry is making a strong bid for serious consideration in the coming years. That language is C, a product of Bell Laboratories, the research arm of AT&T.

Though C has developed a vocal and growing following among minicomputer users – particularly those working in the Unix operating system environment – it has only recently become available on microcomputers.

The specification for C was published in *The C Programming Language* by Kernighan and Ritchie (published by Prentice-Hall, 228 pages). Any serious C programmer will eventually want to study it, but it includes only one chapter of tutorial introduction. Its terse, somewhat scholarly style is probably not appropriate for many microcomputer users, who have widely varying backgrounds and might want more introductory material.

C Programming Guide addresses this new group of potential C programmers. Purdum is the author of an implementation of C for systems using the Z80 CPU. Though the *Guide* could serve as an introduction to any version of standard C, its style and approach are similar to other popular microcomputer programming tutorials.

In nine chapters, Purdum covers the syntax and use of C in an orderly, conversational manner using numerous example programs and program fragments. Because of its extensive set of operators and data types, learning C can be a challenge to those who have only used BASIC or For tran. But Purdum's incremental approach minimizes the difficulty, stressing hands-on experience of C and occasionally offering side-by-side comparisons of similar programs in C and BASIC.

Most of the programs included in the text should run on any C compiler that is Unix version-7 compatible, so the reader with access to such a compiler can get immediate feedback by entering and testing the programs.

Sometimes the text raises questions that can only be answered by running a particular program. The student is expected to enter, compile, and run the program and observe its results. This approach may limit the usefulness of the book for those without access to a compiler. The preface mentions "two underlying assumptions: (1) the only way to learn a language is to write programs with it, and (2) learning is made easier if you can visualize what a statement does," so perhaps that was the author's intention.

While Purdum often suggests "playing around" with a particular statement or function, specific exercises (and solutions to selected exercises) would also be helpful. The student would then have the option of choosing the intuitive, exploratory approach or a more orderly, disciplined one.

Despite these concerns, the *Guide* is a worthwhile tutorial introduction to a powerful and versatile language. C has been called a "mid-level" language, offering the direct control over machine architecture which we normally associate with assembly language, while including a range of constructs and operations typical of high-level languages. Execution speed is comparable to assembler, with the important bonus that C was designed to be portable. Unlike assembly language programs, a C source file can be recompiled to run on a variety of CPUs, generally with little or no change required. These features help make C an attractive language for systems programming – a fact that has not been lost on some of the biggest software houses, who are reportedly writing more and more of their programming languages and utilities in C.

In general, C programs consist of a series of functions, any of which may be placed into libraries and used again in other programs. The capability to develop and test a function only once, then use it freely without the need to modify or even test it, is one of the most attractive aspects of C. The text thoroughly and clearly explains the nature and use of functions, and how they are combined into complete programs.

The index is thorough and easy to use, and appendices contain a syntax summary of C and a list of commercially available C compilers for microcomputers. Along with publishers' names, addresses, and retail prices, Purdum offers brief comments including the degree of compliance with the Kernighan and Ritchie standard. To his credit, Purdum avoids making his book an advertisement for Eco-C – his own version of the compiler. In fact, his association with that particular C compiler is not explicitly mentioned, and a comment describes Eco-C only as "a very nice implementation."

For all its importance to minicomputer users, there are surprisingly few tutorial introductions to programming in C. Though *C Programming Guide* is no substitute for Kernighan and Ritchie's text (nor was it intended to be), it may serve to introduce the microcomputer user to a powerful programming language, one likely to assume an increasingly important position in the microcomputer world.

Microcomputer Graphics Techniques and Applications

By Donald Hearn and Pauline Baker
Published by Prentice-Hall, Inc.
$24.94 cloth or $19.95 paper, 302 pages
Reviewed by Robert Ashworth

This textbook is divided into fourteen major chapters dealing with almost everything a person would want to know about computer graphics on a small computer. Sixteen color figures showing the latest shading techniques as well as many black and white illustrations make this an informative and enjoyable book to read, study, and even give to a friend for browsing.

Hearn and Baker divide the book into five sections. The first four present an overview, the fundamentals of microcomputer graphics, and details on special effects and 3D. The fifth section contains program design, special techniques, simulations, computer-assisted instruction, budget nutrition charts, and game playing. The 92 BASIC programs progress from a simple nine-line picture of of a fish to a 180-line, three-page program. Each chapter concludes with several challenging projects for further programming.

The approach is straightforward, logical, and informative. The graphics command conversion table in the appendix is very useful; it gives the equivalent graphics commands for the Apple, Radio Shack, and the IBM PC. This is an enjoyable

book to read and study, and it offers many varied illustrations to excite the eye and challenge the mind.

Introduction to Numerical Computation in Pascal

by P. M. Dew and K. R. James
Published by Springer-Verlag, New York, Inc.
$16.00, 291 pages
Reviewed by Robert Ashworth

The first part of this book focuses on Pascal with an emphasis on the fundamental techniques needed to exploit the modular structure of the language. The authors have developed very readable code with particularly clear descriptions to show the results of quality mathematical software.

The central theme is a description of "mathlib" which is a collection of routines available from the Editor of the Computer Science series of the publisher. *These run under* UCSD Pascal on the Apple II microcomputer. A ten-page summary of all the routines is given in the appendix.

The authors take an algorithm or procedure for the solution of a given problem and then show, step-by-step, how to make it more comprehensive, accurate, or efficient. Stress is placed on: stability, i.e., the algorithm is not sensitive to rounding errors; accuracy, i.e., the algorithm performs according to the documentation; efficiency, i.e., the algorithm is a function of time and memory required; robustness, i.e., the algorithm solves correctly the problem it was intended for; adaptability, i.e., the ability to move a program from one computer to another with a minimum of modification; error control, i.e., the need to have results correct to a specified number of decimal places; and the respective mathematical method of analysis. Part I concludes with a detailed discussion of errors and how to manage them on your computer.

Part II explores the development of mathematical software. Different methods are given for the solution of nonlinear equations in one variable. The study of systems of linear equations is followed by different methods and an analysis of the strengths of each approach. Finally, the last two chapters deal with the fixed-point rules and the adaptive methods for numerical integration.

The text meets the authors' goal of covering core material in numerical analysis and methods for producing reliable mathematical software. Solutions are given for all exercises and four helpful appendices complement the well-designed text. It should be given a high priority for Pascal programmers.

Automation of Reasoning

Classical papers on computational logic, Vol. 1, 1957-1966; Vol. 2, 1967-1970
Edited by Jorg Siekmann and Graham Wilson
Published by Springer-Verlag: Berlin, Heidelberg, New York, 1983
$35.00 Vol. 1: 525 pages, Vol. 2: 637 pages.
Reviewed by Jay Halcomb

These are the first two volumes of a planned series devoted to automated theorem proving. The volumes collectively contain 61 articles discussing mechanical theorem proving, which has applications in automated program verification, program synthesis, and deductive data bases. The papers are of the highest quality, selected by an international committee of researchers in the field, and include translations of previously untranslated literature.

On the whole the papers are focused in scope, approaching computational logic in its strictest sense: the application of theories of first-order deduction and semantic methods to proof of mathematical or logical theorems. A few of the papers allude to possible wider applications, such as generalized problem solving and program verification, but the core concern is with theorem proving in logic and mathematics. In this sense the title and cover blurb are slightly misleading. However, for readers with professional or avocational interests in this field, the book is an excellent unified reference source.

The volumes contain three historical articles especially useful to the new student or casual onlooker:

Wos and Henkin: "Automated thoerem proving, 1965-70"

Davis: "The prehistory and early history of automated deduction"

Maslov, Mints, and Orevkov: "Mechanical proof search and the theory of logical deduction in the USSR"

Background

The philosopher G. Leibnitz was perhaps the first to seriously envisage the possibility of algorithmizing and even mechanizing human reasoning. He proposed a *calculus ratiocinator* and *lingua characteristica* "to bring under mathematical laws human reasoning, which is the most excellent and useful thing we have . . . the mind will be freed from having to think directly of things themselves, and yet everything will turn out correctly." But this vision lacked foundations until the growth of mathematical logic through the work of Boole, Frege, Russell, and Whitehead and the creation of recursion theory by Turing, Church, and Kleene.

The subsequent implementation of these mathematico-logical theories in actual machines in this century brought the matter to a practical head. But until very lately enthusiasm and optimistic speculation have been more characteristic of the field than have actual results. However, the recent announcement of a proof by computer of the famous four-color problem, a proof that apparently was not to be achieved in any other manner, established a substantial and important mathematical theorem, and the automation of human reasoning began to come of age.

Work on theorem proving in recent decades first concentrated, naturally perhaps, on the verification of known results, testing theories of problem solving rather than attempting new results. Thus in the late 50s the logician Hao Wang was able to deduce all of the theorems of Russell and Whitehead's *Principia Mathematica* in a few minutes on an IBM computer. Later workers began attempting to formulate new results in the areas of group theory and model theory and achieved some noteworthy successes, establishing previously unknown theorems of some significance. But the crowning feat to date remains the aforementioned mechanical solution of the famous four-color problem.

Briefly, the four-color problem questions whether four colors will always suffice for coloring any map on a two-dimensional surface. Much work was expended on this deceptively simple problem during the past century, but the general case remained elusive until the announcement in 1976 of a computer-assisted solution by Appel and Haken ("Every planar map is 4-colorable," *Bull Am Math Soc*, No. 82, 1976). It should be noted, however, that this success relied on a prior ingenious reduction of the problem to a large but finite number of cases, which the machine searched as the "proof," so the result is not unalloyed. This work is not discussed in the present volumes, due to its more recent date, but will doubtless appear in the next of the series.

Schools of Automated Reasoning

There are two general approaches to the simulation of reasoning by computers. One is the implementation of strictly deductive theory for a specific class of problems (such methods achieved the results mentioned above). The other is a heuristic, generalized, problem solving approach that requires the computer to generate "reasonable" search directions in problem solving, in this way emulating the human mind. This approach is associated with the work of Newell, Simon, and Shaw, early pioneers in artificial intelli-

(Continued on page 101)

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

PC-DOS Close Function

I was tipped off to the first topic for this month's column by an old COBOL programmer who had been trained to close every file in sight if execution of his program was terminated due to some error condition. When he tried to persist in this habit under PC-DOS, his files began to mysteriously disappear. They would still appear on a directory listing, but the file lengths were displayed as zero bytes and the CHKDSK utility was finding all sorts of "lost clusters" that it had to clean up.

The problem turned out to be that PC-DOS is not checking to see if a file control block has been properly activated by a previous successful OPEN or MAKE call before carrying out the CLOSE operation. Apparently it just looks up the corresponding entry in the disk directory and copies in the zeroed-out file control block information, effectively destroying it. PC-DOS then gives the calling program back a return code indicating success (AL=zero). This is a really surprising bug, since there are several ways that the operating system can quickly assess whether a file control block is valid or not.

I have vivid memories of the reception given to the poor soul who originally reported the IBM PC BASIC math routine errors, and also of the rather indignant letter previously directed to this column by the grandfather of PC-DOS 1.1 (Tim Paterson). Therefore, I am supplying a detailed example program in Listing One (page 85) that can be used to verify that this problem exists in both PC-DOS 1.1 and 2.0. I haven't had time yet to explore what happens when you ask for other disk operations (such as read or write record) with an unopened file control block, but I'm sure it will turn out to be most amusing.

The Microsoft Assembler

The Macro Assembler for the IBM PC, which was supplied by Microsoft, has come in for a little criticism from time to time because of its lack of speed and its appetite for memory, but it has escaped otherwise unscathed in all the magazines I read. Its performance and size can be attributed to the fact that it was written in Pascal (a more unlikely high level langauge in which to write such a systems tool can hardly be imagined), and in spite of its handicaps it has been in the Softalk Top 30 month after month.

Certain "features" of the Assembler make me conclude that it was written to a set of specifications by a hotshot bunch of Pascal hackers who never even tested it, let alone used it for any kind of job in the real world. The first evidence for this is found in some bizarre listing notations. Although every assembler I have encountered will show you the real value of an equate in the object code column of the listing, the Microsoft Assembler gives you a signed hexadecimal integer (see Listing Two on page 87, line 15).

The authors of the Assembler also chose to ignore the byte-swapped nature of 16-bit values on Intel processors in their listing format. In line 21 of Listing Two, the byte at address 0000 appears to contain 00 while the byte at 0001 contains 01 — of course the reverse is what is truly found in the object code. Patching a program working from such a listing is unnecessarily confusing.

But as long as we're ragging on the boys at Microsoft, how about pointing out some really gross bugs.

The SHL and SHR operators are not mentioned at all in the IBM version of the Assembler manual, except in a brief example on page 4-21 which states that 101B SHL (2*2) should yield 01010000B, or 50H. From this you can infer that the SHL operator is used in the form "data SHL shift_count," which in fact agrees with the Intel specification. Unfortunately, no one ever checked the actual operation of the Assembler against the example in the manual. By experiment, we find that the Microsoft Assembler expects the shift count first (see lines 29-37 of Listing Two), in conflict with both the Intel and Digital Research 8086 assemblers.

The SHR operator, on the other hand, behaves entirely unpredictably. In some cases it seems to act like the SHL operator, in others it doesn't return any understandable result at all (see lines 39-53).

Intel defines SHR and SHL as performing "logical shifts," bringing zero bits into the vacated positions. The Microsoft Assembler is inconsistent on this point (see lines 57-62). My guess is that the authors are ANDing the shift count with a mask of 0FH, instead of testing for a shift count greater than 15 which should return a zero result for any data.

The logical operators EQ, NE, GT, GE, LT, and LE all perform erroneously as often as not (see lines 64-91). I surmise here that the operators are somehow ignoring the signs of decimal integers. Other than that, the operators apparently treat their arguments as *unsigned* 16-bit integers, a fact that is not documented in the manual but can be deduced from comparing hex values (for example, 07FFFH LT 08000H returns TRUE).

The Not operator works for positive decimal integers and all hexadecimal integers, but fails with strange results for negative decimal integers (see line 95). Similarly, the logical Inclusive Or operator gives incorrect results with negative decimal integers.

Lastly, the Exclusive Or operator appears to work as an Inclusive Or instead (see lines 107-110). Don't imagine for a moment that we have covered all the bugs! These were located in just a few minutes' work, and I have been tipped off to others that should be good for several more *DDJ* columns.

The moral of this month's story seems to be: Avoid use of exotic or complex expressions, and perform any calculations involving logical operations at runtime rather than assembly time. I had to laugh when I found the note on page D-14 of the Assembler manual that said, "In general, be careful when patching in hex." Usually, this would be advice too obvious and banal to be worth the ink, as though someone were trying to say, "In general, be careful when jumping off a cliff." But in this case, you are probably at least as safe patching in hex as you are in using the Assembler.

Interesting side observations: Both the Intel assembler ASM86 and the Digital Research assembler RASM86 performed all of the operations in Listing Two correctly. They also take up about half as much space on the disk as the Microsoft Assembler and run much faster. RASM86 is now available for PC-DOS as part of the Digital Research "Programmer's Utilities" package.

Highly Recommended Software

Those of you who like RAM-disks (and who doesn't?) should consider the purchase of JETDRIVE for DOS 2.0 from Tall Tree Systems in Palo Alto, California. The program is solid as a rock, works with any RAM expansion board, costs only $40, and is even supplied with source code. Also included is the JET utility which transfers files between disks at a rate guaranteed to astonish the most jaded programmer. If you thought the PC-DOS COPY function was fast com-

pared to CP/M's PIP, you won't believe JET.

And for those small software houses wrestling with the problem of how to supply all the eighty zillion 5¼-inch, soft-sectored disk formats, inexpensive relief is at hand. The program UNIFORM, from Micro Solutions in DeKalb, Illinois, can initialize disks and copy files to or from disks for some 40 different computers including such formerly difficult animals as the QX-10 and the Otrona. It will also transfer files between a CP/M disk and any of the MS-DOS single- or double-sided formats. UNIFORM currently runs on the various Kaypro models but is being ported to other machines. The best news of all is that UNIFORM costs only $50! At that price, it's worthwhile to buy a Kaypro just to use it – the alternatives are hideously expensive. **DDJ**

16-Bit Toolbox

Listing One

```
                                        name    optest
                                        page    55,132
                                        title   'OPTEST - test Microsoft Assembler operators'
                                ;
                                ; Show operation of various operators and demonstrate some
                                ; notational idiosyncracies in the Microsoft IBM PC Assembler.
                                ;
                                ; Ray Duncan, November 1983
                                ;
                                ;
                                ; Every other assembler I have ever encountered will display
                                ; the true hex equivalent of an equate in the object code column.
                                ; The Microsoft Assembler, however, shows a signed hex integer!
                                ;
=-0001                          neg_one equ     -1      ;should display as FFFF


                                ; For unknown reasons, the Microsoft Assembler also fails to display
                                ; the byte swapped nature of some 16 bit values.

0000  0001                              dw      1


                                ; The Microsoft Assembler manual says nothing about the SHR
                                ; and SHL operators.  However, an example on page 4-21 states
                                ; that the operation 101b shl (2*2) should return 01010000b or 50H,
                                ; implying that the order of arguments is data SHL shift_count.

0002  0080                              dw      101b shl (2*2)

                                ; Since Microsoft's own example doesn't work with their assembler,
                                ; by experimenting we find that the expected order of arguments is
                                ; shift_count SHL data. This conflicts with the Intel specification.

0004  0001                              dw      0 shl 1
0006  0002                              dw      1 shl 1
0008  0004                              dw      2 shl 1

                                ; The SHR operator doesn't work correctly.  Apparently gives
                                ; the same results as SHL...

000A  0004                              dw      2 shl 1
000C  0004                              dw      2 shr 1

```

(Continued on next page)

Listing One

```
                                ; Except when it gives no result at all...

000E  0010                            dw      1 shl 8
0010  0000                            dw      1 shr 8

                                ; SHR may even give different results with equivalent data

0012  FFFF                            dw      16 shr -1
0014  0000                            dw      16 shr 0ffffh

                                ; Sometimes the SHL operator seems to perform a "logical shift"

0016  FFFE                            dw      1 shl -1
0018  8000                            dw      15 shl -1

                                ; other times, it appears to perform a circular shift

001A  FFFF                            dw      16 shl -1

                                ; The EQ operator doesn't work properly

001C  FFFF                            dw      1 eq 1
001E  FFFF                            dw      1 eq -1

                                ; The EQ operator can give different results with equivalent data

0020  FFFF                            dw      1 eq -1
0022  0000                            dw      1 eq 0ffffh

                                ; The NE operator is similarly afflicted

0024  0000                            dw      1 ne 1
0026  0000                            dw      1 ne -1
0028  FFFF                            dw      1 ne 0ffffh

                                ; The LE, LT, GE, and GT operators give confusing results

002A  0000                            dw      -1 lt 1
002C  FFFF                            dw      -1 le 1
002E  0000                            dw      -1 gt 1
0030  FFFF                            dw      -1 ge 1

                                ; Again, these operators can give different results with
                                ; equivalent data

0032  FFFF                            dw      1 ge -1
0034  0000                            dw      1 ge 0ffffh

                                ; The NOT operator fails miserably on some signed integers

0036  0002                            dw      not -1
```

```
                                ; Similarly, the OR operator flubs with signed integers

0038  0001                              dw      -1 or 0
003A  FFFF                              dw      0ffffh or 0

                                ; The XOR operator apparently works as an Inclusive OR
                                ; instead of Exclusive OR

003C  0000                              dw      0 xor 0
003E  0001                              dw      1 xor 0
0040  0001                              dw      0 xor 1
0042  0001                              dw      1 xor 1

                                        end
```

End Listing One

Listing Two

```
                                        name    closer
                                        page    55,132
                                        title   'CLOSER - show bug in PC-DOS function 10H'
                                ;
                                ; This program demonstrates a subtle but dangerous bug in the
                                ; PC-DOS Close File function 10H.  If a Close request is issued
                                ; using a file control block that has not been previously
                                ; activated by a successful Open command, the file's length
                                ; will be truncated to zero and the clusters previously assigned
                                ; to the file are left floating.
                                ;
                                ; Ray Duncan, November 1983

= 000D                          cr      equ     0dh     ;ASCII carriage return
= 000A                          lf      equ     0ah     ;ASCII line feed

0000                            cseg    segment para public 'CODE'

                                        assume  cs:cseg,ds:data,es:data,ss:stack

0000                            closer  proc    far
0000  1E                                push    ds      ;save DS:0000 for final
0001  33 C0                             xor     ax,ax   ;return to PC-DOS
0003  50                                push    ax

0004  B8  ---- R                        mov     ax,data ;make our data area
0007  8E D8                             mov     ds,ax   ;addressable
0009  8E C0                             mov     es,ax

                                                        ;now create file QUACK.DAT
000B  B4 16                             mov     ah,16h
000D  BA 013A R                         mov     dx,offset fcb
0010  CD 21                             int     21h
0012  0A C0                             or      al,al   ;create successful?
0014  75 4D                             jnz     closer8 ;no,jump
```

(Continued on next page)

16-Bit Toolbox (Listing continued, text begins on page 84)

Listing Two

```
36      0016  B4 09                     mov     ah,9        ;yes,print success message
37      0018  BA 0000 R                 mov     dx,offset msg1
38      001B  CD 21                     int     21h
39
40                                                          ;now set the record length
41                                                          ;to 1024 bytes and write
42                                                          ;random data into the
43                                                          ;file (using default DTA)
44      001D  C7 06 0148 R 0400         mov     word ptr fcb+14,1024
45      0023  B4 15                     mov     ah,15h
46      0025  BA 013A R                 mov     dx,offset fcb
47      0028  CD 21                     int     21h
48      002A  0A C0                     or      al,al       ;was write successful?
49      002C  75 35                     jnz     closer8     ;no,jump
50      002E  B4 09                     mov     ah,9        ;yes,print success message
51      0030  BA 001B R                 mov     dx,offset msg2
52      0033  CD 21                     int     21h
53
54                                                          ;now close the file so the
55                                                          ;directory will be updated
56      0035  B4 10                     mov     ah,10h
57      0037  BA 013A R                 mov     dx,offset fcb
58      003A  CD 21                     int     21h
59      003C  0A C0                     or      al,al       ;close operation successful?
60      003E  75 23                     jnz     closer8     ;no,jump
61      0040  B4 09                     mov     ah,9        ;yes,print success message
62      0042  BA 0041 R                 mov     dx,offset msg3
63      0045  CD 21                     int     21h
64
65                                                          ;now clear out the file control
66                                                          ;block except for the filespec,
67                                                          ;as though the file had never
68                                                          ;been opened, and then request
69                                                          ;another close operation.
70
71      0047  BF 0146 R                 mov     di,offset fcb+12
72      004A  B9 0019                   mov     cx,25
73      004D  32 C0                     xor     al,al
74      004F  FC                        cld
75      0050  F3/ AA                    rep stosb           ;this zeros out the fcb
76
77      0052  B4 10                     mov     ah,10h      ;now close file again
78      0054  BA 013A R                 mov     dx,offset fcb
79      0057  CD 21                     int     21h
80      0059  0A C0                     or      al,al       ;check status
81      005B  75 06                     jnz     closer8     ;bad status,jump
82
83                                                          ;status ok, print final
84                                                          ;message to inspect directory
85      005D  BA 0064 R                 mov     dx,offset msg4
86      0060  EB 04 90                  jmp     closer9
```

```
                                ;come here if unexpected
                                ;failure of disk operation
0063 BA 0111 R          mov     dx,offset msg5

                                ;print message and exit
0066 B4 09              mov     ah,9
0068 CD 21              int     21h

006A CB                 ret                     ;far return to PC-DOS

006B                    closer  endp

006B                    cseg    ends


0000                    stack   segment         para stack 'STACK'
0000    40 [                    db              64 dup (?)
                ??
                    ]

0040                    stack   ends


0000                    data    segment         para public 'DATA'

0000 0D 0A              msg1    db              cr,lf
0002 46 69 6C 65 20 51          db              'File QUACK.DAT created'
     55 41 43 4B 2E 44
     41 54 20 63 72 65
     61 74 65 64
0018 0D 0A 24                   db              cr,lf,'$'

001B 0D 0A              msg2    db              cr,lf
001D 31 30 32 34 20 62          db              '1024 bytes written into QUACK.DAT'
     79 74 65 73 20 77
     72 69 74 74 65 6E
     20 69 6E 74 6F 20
     51 55 41 43 4B 2E
     44 41 54
003E 0D 0A 24                   db              cr,lf,'$'

0041 0D 0A              msg3    db              cr,lf
0043 46 69 6C 65 20 51          db              'File QUACK.DAT closed properly'
     55 41 43 4B 2E 44
     41 54 20 63 6C 6F
     73 65 64 20 70 72
     6F 70 65 72 6C 79
0061 0D 0A 24                   db              cr,lf,'$'

0064 0D 0A 0A           msg4    db              cr,lf,lf
0067 53 65 63 6F 6E 64          db              'Second close operation requested
     20 63 6C 6F 73 65
```

(Continued on next page)

Listing Two

```
140           20 6F 70 65 72 61
141           74 69 6F 6E 20 72
142           65 71 75 65 73 74
143           65 64 20
144     0088  6F 6E 20 66 69 6C                 db        'on file QUACK.DAT'
145           65 20 51 55 41 43
146           4B 2E 44 41 54
147     0099  0D 0A                             db        cr,lf
148     009B  49 6E 73 70 65 63                 db        'Inspect length of file QUACK.DAT '
149           74 20 6C 65 6E 67
150           74 68 20 6F 66 20
151           66 69 6C 65 20 51
152           55 41 43 4B 2E 44
153           41 54 20
154     00BC  6F 6E 20 64 69 72                 db        'on directory listing'
155           65 63 74 6F 72 79
156           20 6C 69 73 74 69
157           6E 67
158     00D0  0D 0A                             db        cr,lf
159     00D2  74 68 65 6E 20 72                 db        'then run CHKDSK with /F switch to
160           75 6E 20 43 48 4B
161           44 53 4B 20 77 69
162           74 68 20 2F 46 20
163           73 77 69 74 63 68
164           20 74 6F 20
165     00F4  72 65 63 6F 76 65                 db        'recover lost disk clusters'
166           72 20 6C 6F 73 74
167           20 64 69 73 6B 20
168           63 6C 75 73 74 65
169           72 73
170     010E  0D 0A 24                          db        cr,lf,'$'
171
172     0111  0D 0A                   msg5      db        cr,lf
173     0113  55 6E 65 78 70 65                 db        'Unexpected failure of disk operation'
174           63 74 65 64 20 66
175           61 69 6C 75 72 65
176           20 6F 66 20 64 69
177           73 6B 20 6F 70 65
178           72 61 74 69 6F 6E
179     0137  0D 0A 24                          db        cr,lf,'$'
180
181                                                     ;file control block
182     013A  00                      fcb       db        0         ;use default drive
183     013B  51 55 41 43 4B 20                 db        'QUACK   DAT'
184           20 20 44 41 54
185     0146     19 [                           db        25 dup (0)
186                  00
187                     ]
188
189
190     015F                          data      ends
191
192                                             end       closer
```

End Listing Two

C/UNIX PROGRAMMER'S NOTEBOOK

by Anthony Skjellum

In the first installment of this column, I proposed a standard for the layout of C code (see *DDJ* No. 84, October 1983). Significant reader response was received concerning this subject, the vast majority of which was in favor of the concept of a C layout standard. In this column, I will present reader comments concerning the layout standard and discuss some modifications and additions to the proposed standard based on reader suggestions.

Some comments were also received concerning my discussion of runtime library and linkage format incompatibilities. Discussion of these points will be left for a future column.

Questions of White Space

Several readers took exception to a particular point in the proposed layout standard, 4F, which states, "No white space character is placed between a keyword (e.g., *if*) and its parenthesized argument." David D. Clark of State College, Pennsylvania, writes:

"In general, I like your coding standard suggestions. My only strong objection is to your idea to leave out spaces between reserved words [and their arguments]. It makes them look like function invocations."

Tim Smith of Evanston, Illinois, notes:

"... I think that a single space between a function name and the initial opening parenthesis, or after '*if*s' and '*else*s,' looks better ...."

Guy Scharf of Mountain View, California, writes:

"4f. I have a strong preference to *always* put a space between a reserved word (e.g., *if, while*) and its parenthesized argument. This adds legibility for me."

Finally, Charlie Brady of New South Wales, Australia, writes:

"The only real beef I have with you is the formatting of keywords and their parenthetical expressions. I can see no reason to depart from Kernighan and Ritchie on this point, and a number of reasons for maintaining their convention. Firstly, a flow control construct is semantically distinct from a function call, and a formatting difference is a reasonable way of distinguishing them. Secondly, the formatting difference simplifies the use of a text editor for such tasks as constructing structure charts. Third, your recommendation departs from at least three extant recommended standards, namely Kernighan and Ritchie, Thomas Plum (*C Standards and Guidelines*, Plum-Hall, 1981) and Tim Lang ('Formatting C,' *AUUGN*, Vol. 4, No. 1, Jan. 1982. Enclosed)." (I want to thank Mr. Brady for including a copy of the Lang article with his letter. The C standard proposed there is very compatible with the one I have proposed.)

After considering the above remarks, I have come to the conclusion that the space really does serve a useful purpose. Therefore, I suggest that point 4f should be changed to read: "A single white character is (optionally) placed between a keyword (e.g., *if*) and its parenthesized argument." (Making the white space character optional is another point for debate.) I think it should be optional but recommended. I don't think that adding a space for function call invocations would be beneficial, as suggested by Mr. Smith.

Another question concerning white space insertion comes in connection with argument lists. The original standard does

```
(a)  if( (a == 1) && (b == 2) && (c == 3) && ( (d == 4) || (d == 5) ) )
     {
          /* operations performed if conditional true */
          ...
     }

(b)  if     /* we make multi-line expression look like a block */
     (
          (a == 1) &&
          (b == 2) &&
          (c == 3) &&
          ( (d == 4) || (d == 5) )
     )
     {
          /* operations performed if conditional true */
          ...
     }
```

Figure 1.

(Under section 4)

b. Binary operators (e.g., +, -, /, but not -> and .) and assignment operators (e.g., =, *=, and &=) are delimited by white space.

g. Parentheses should be adjacent to the argument(s) which they enclose.

h. A comma is bound to the argument which precedes and should be followed by a single space.

i. Operators such as -> and . (used in pointer references) directly bind to their arguments with no intervening spaces.

Table 1.

not indicate if spaces should be included. It is my opinion that a comma should be directly adjacent to the argument that it follows, and that a single white space should follow each comma to add legibility. I am also convinced that parentheses *should* be adjacent to the argument(s) they enclose. Thus (in agreement with Tim Lang's article mentioned above), I would write:

```
x = atan(sin(y));
```

and not

```
x = atan( sin( y ) );
```

Yet another point not previously mentioned is that binary operators should be delimited by white space. Thus, the following statement lacks sufficient white space:

```
v = sin(1n(1.0+x));
```

while this expression is properly formed:

```
v = sin(1n(1.0 + x));
```

Finally, section 4 needs to be updated to include a style specification for pointer references. I think that operators " . " and "—>" should not be delimited by spaces from the objects which they act on. This point and the three above are formalized in Table 1 (page 94) as additions to section 4.

The Lang article points out a circumstance under which point 4g need not be followed. This occurs when very complicated conditional expressions of the form "**keyword** (expr)" are split over several lines. In Figure 1 (page 94), for example, instead of using a crowded expression (Figure 1a), a more readable form is selected (Figure 1b). Note that in Figure 1b *the* parentheses are placed on lines by themselves, since they bracket a multiline expression, much like braces enclose the statements of a block.

Another point of minor objection was the tabulation method specified by the standard (point 1a). Steve Newberry of Los Altos, California, states:

"Upon one point I do feel compelled to argue with you, and that is the tab convention: The *depth* of the tab stop on a given page is of far less significance to the readability of that page than is the *consistency* of the depth. I *really* don't want to use different size tabs on the same page."

Tim Smith writes:

"I personally follow most all of his suggestions on how to actually format the code on the line and page, with only two exceptions. I *always* use 4-space tab stops . . . . "

I agree that having a single tab size is the preferable way to write C code. Standard tabs give more openness to the code, and make various parts of a program easier to pick out. My rationale for large horizontal tabbing is the same as for vertical tabbing. I want the program's significant portions to stand out. However, I propose adding point 1b to the standard: "4-space tabs may be used in lieu of standard tabs in cases where a subprogram includes highly nested segments." I would also include point 1c: "Only one of the two tabbing conventions should be employed in any given program module."

Some currently available screen editors provide a feature called horizontal scrolling With horizontal scrolling, the user views a window of the file in both the vertical and horizontal directions. Thus, files with lines longer than the display device may be handled intelligently. Under such circumstances, there is no real disadvantage to using standard tabs to any desired nesting depth, which is permitted under point 1a of the proposed standard.

Other Corrections

Charlie Brady noted an unnecessary point in section 3e which stated: "When a null block is used (e.g., '{}'), it may appear on the same line as other statements (e.g., do {} while(expr);)." He writes:

"Another minor point of disagreement concerns the use of the null block ({}). This is never necessary, and I believe that the null statement (;) is clearer. It should be emphasized that the null statement *deserves* a line of its own. Your example:

```
do {} while(expr);
```

is more simply written

```
while (expr)
    ;
```

"

0. Identification Description (I.D.) information must appear at the beginning of each C language source file.

   a. The recommended format is:

      i. Begin comment (/*).
      ii. Space.
      iii. Title (identification name normally = filename).
      * iv. Sub-title (i.e., what program system does this file belong to).
      v. Space.
      vi. Classification (see below).
      vii. Year.
      viii. Owner.
      ix. Status (see below).
      * x. Current Version number and brief history.
      xi. Date.
      xii. Functional/Structural Description in brief.
      * xiii. Portability synopsis.
      * xiv. Space.
      xv. End Comment (*/).
      * xvi. Space.

   b. The program classification (vi.) is one of the following:

      i. Public-domain.
      ii. Copyright.
      * iii. Copyright: released for non-commercial purposes.
      iv. Unclassified.
      v. Secret.
      vi. No classification.

   c. The program status (ix.) is one of the following:

      i. Outline.
      ii. Draft.
      iii. Test (alpha, beta, etc.).
      iv. Release.

Table 2.

In accordance with Mr. Brady's remark, I propose replacing point 3e with the following: "The null block ({ }) can always be avoided. Instead of a null block, use the null statement." I also would add point 3f ii: "The null statement is always on a line by itself."

Documentation Standards

In addition to a code layout standard, James Halstead of Joliet, Illinois, has proposed a basic documentation standard. He writes:

". . . I strongly suggest that the original standards one through nine be renumbered two through ten so that the first and foremost standard may be inserted."

The documentation standard suggested by Mr. Halstead is presented in Table 2 (page 96) and I think it should be included in my proposed standard. However, I have decided to number the documentation standard as Section 0 in order to preserve the numbering of the current sections. (The slight additions I have made include an asterisk to indicate the addition.)

In addition to Section 0, I think a basic documentation standard is in order for functions as well. Such a standard is presented in Table 3 (page 96). I have placed this under Section 10, since I think of function documentation as a separate task from module documentation as described in Section 0.

Other Proposals

Several other readers made suggestions for the standard. I think that Tim Smith proposes several which merit discussion. They are presented here with the point numbers they receive as part of the standard:

5c. Don't nest comments, even if your preprocessor/compiler allows it.

6f. If there are many declarations, whether one line or many, alphabetize them.

9g. Restrict variable and function names to seven well-chosen characters, even if your compiler allows more.

Steve Newberry writes the following about standards:

"I applaud your interest in establishing a standard format convention for C programs. However I feel that your effort would have more impact if tied to support of Tom Plum's book, *C Programming Standards and Guidelines, Version U (Unix and offspring)*, Edition 3: Jan. '82. Presented in this manner, your proposed formatting standard would be seen as a consistent extension of a more general set of standards already in wide circulation."

Other Points of View

Although most readers were favorable to the idea of a C format standard, Douglas M. Potter of Seattle, Washington, writes:

"I'm afraid I don't see much advantage of your proposed standard over theirs [Kernighan and Ritchie]. In both cases, the size of the indent is too large. I always run out of room on the right side with a tab-sized indent. I also find that nobody uses enough white space."

John F. Draffen of Texas City, Texas, wrote me a detailed letter on why he didn't like the idea. He writes:

"I am writing to express my objections . . . In the first place, I do not think a standard of this type is either necessary or desireable. The layout has nothing to do with portability which to my mind is the only excuse for a standard. It seems to me that it is hard enough to get people to agree on necessary standards.

"In the second place, I do not agree with many of your suggestions on style. One of the nice things about C that it shares with Fortran is its relative conciseness. I do not like to see code strung out unnecessarily. C does not interject unnecessary constructions, and I think that introducing unnecessary white space, excessive indentation, and meaningless comments is a kind of gingerbread that we can do without." Mr. Draffen's style of philosophy is listed in Table 4 .

I don't really agree with Mr. Draffen on several counts. First and foremost is that portability is not the sole subject of importance in programming. The ability to maintain, understand, correct, and enhance code is of great importance. To understand someone else's code (or your own code at a later date) requires some degree of formatting. Comments which seem less than essential to the programmer must sometimes be included for the sake of others. This is immensely important. It is often difficult for programmers to know how to comment their code, since they usually cannot know the level of sophistication of later readers. Thus, it is often better to include a few extra comments, than to comment code sparsely.

I suggested in my previous column that users should maintain their own code in the form that they prefer. However, code distributed to others could (and should) meet some minimum standard of neatness (i.e., formatting) and presentation. Some of this can be provided by a beautifier but most must be done by the programmer.

As one final note on C layout, I'm including an insightful paragraph which Tim Smith included in his letter. It suggests why so much C code is so poorly formatted and commented:

(Continued on page 101)

10. Each function should contain the following minimum documentation:
    a. A general explanation of the function performed.
    b. Its name, and a description of its arguments including their types, and legal values.
    c. A description of the functional return value, if any.
    d. A list of non-standard functions used by the function.
    e. A list of external variables used and/or modified by the function.
    f. A description of the error handling characteristics of the function.
    g. A valid calling sequence example, if practical.

Table 3.

1. Punctuation should be used sparingly. The insertion of unnecessary white space should be avoided.
2. Block structures should be used, indicated by indentation. Excessive indentation should be avoided.
3. Comments and program code should be separated. Comments on the same line as code should be displaced far enough to the right that they do not obscure the code.
4. Comments should be meaningful. Comments that do no more than repeat what has already been said by the code should be avoided.

Table 4.

OF INTEREST

by Michael Wiesenberg

Networking PCs

PCterminal, from Santa Clara Systems, is an IBM-compatible personal computer with built-in local area network. Use it as an intelligent terminal in an IBM PC network, PCnet, instead of IBM PCs, to increase user stations at one-third the cost. You get monitor, keyboard, 8088, serial and parallel interface, four expansion slots, 64K RAM (expandable to 256), built-in networking, and connection for optional 5.25-inch floppy disk drive, for $1295. It runs either PC-DOS or Santa Clara Systems' version of MS-DOS, SCS-DOS. You can connect up to 16 PCterminals to one IBM PC or XT in a Santa Clara Systems network. PCnet, built into the terminals, lets PCs link with thousands of others through special adapter boards and software. "Remote execution" permits one terminal to run a command on another.

8086 C and Pascal

Whitesmiths has Release 2.2 of its **C** and **Pascal** compilers, adding 8086 architecture and Unix-style libraries. Whitesmiths now supports over 30 operating systems on five architectures: PDP-11 family, 8080 and Z80, MC68000, VAX family, and 8086. The last includes 8088, with optional 8087 math coprocessor support plus 80186 and 80286 instruction sets. On IBM PC, programs run under CP/M-86, DOS 1.x, and DOS 2.0. C native compiler is $550, Pascal native $1100, C cross compiler $1100, Pascal cross $1400.

The Digital Doctor

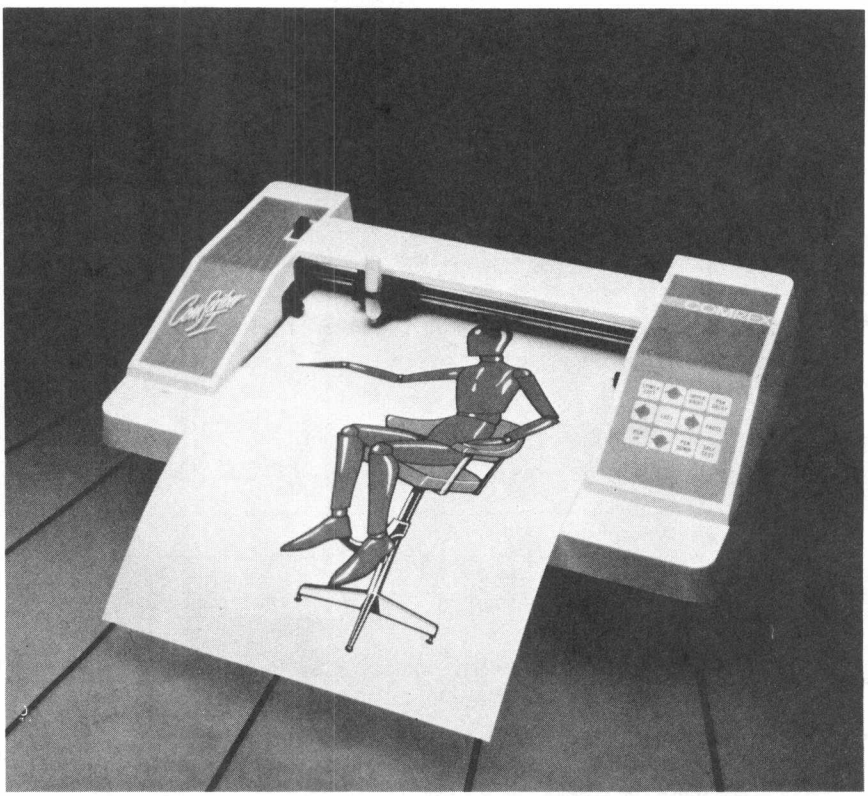
Not to be confused with the resident intern, the digital doctor makes house calls. Data Encore, a subsidiary of Verbatim, offers the **Datalife Disk Drive Analyzer**, a diagnostic software tool to check disk drive performance, testing head alignment, disk clamping, write/read accuracy, and disk speed. $39.95 for IBM PC, XT, and Apple. The latter includes enhanced graphics. When you return the warranty card, Data Encore will send you a free Datalife Twin Pack of minidisks or head cleaning kit.

Box It

You can make your IBM PC portable, or at least somewhat transportable, by putting it in a **Firebilt** case, for $181, and put the monitor and a printer in another case for $191. These are foam-padded luggage style cases made of high-impact plastic. Other cases made of plastic, aluminum, or fiberglass are available for scores of other computers, monitors, and printers, ranging in price from $65 to $315.

Computer Becomes Typewriter

Last time I described a product that turns electronic typewriters into printers. This time, to be fair, I mention a product that lets your printer become a typewriter, controlled by the keyboard of your computer. Hall Associates think that you do not want to be so inefficient as to call up your word processor just to type out one mailing label or a short note, and so they offer **TYPWTR** that lets your



Inexpensive Plotter

The **Model CR-1810 Comscriber I**, from Comrex International, is a lightweight, single-pen plotter that takes paper 8.5 inches wide and up to 10 feet long, and pauses to change pens. The price is $695.

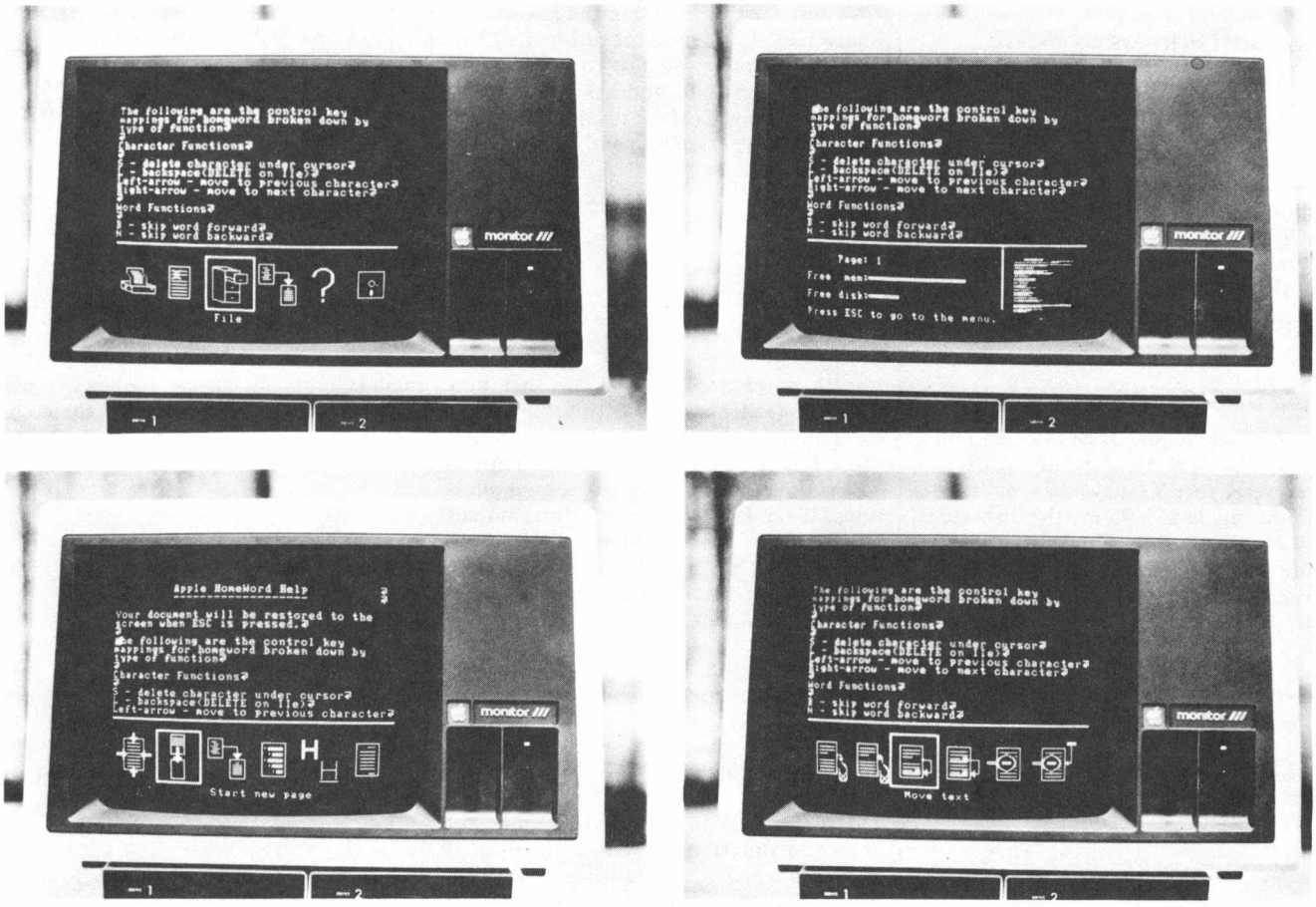
printer immediately track every keystroke. (What's wrong with using CP/M's Control-P?) You can have lines of 250 characters and word wrap; you can set margins and tabs. You need CP/M 2.x, 48K, and a printer that backspaces and does carriage returns without line feed. $29.95 plus $1.50 p. and h.

2068 and Adam Too

Looks like Timex has another winner with its 2068 Personal Color Computer. The under-$200 computer seems to address itself to all the difficulties users of the 1000 and 1500 had. The new machine has a "real" keyboard, not just sound but synthesizer capabilities, instant-load cartridges, high-resolution color graphics, joysticks, etc. What it lacks is a lot of software, although I understand Timex is adapting Sinclair Spectrum software as fast as they can. In the meantime, Softsync fills the gap with several games, **Cosmic Gorilla** ("masses of mutant mohair relentlessly stealing everything they can get their hands on"), **Gulpman** ("The cursed wormoids are out to get control of Gulpland, chasing its inhabitants out of their apple orchards. Eat as many apples as you can to get bonus points and use your lasers to stun the wormoids"), **Cyberzone**, a voice-actuated game, all on cassette and all $19.95. **Voice Chess** talks to you during play, recommending moves, if you wish, letting you switch sides, and analyzing moves, $24.95. Softsync also has **Zeus Monitor/Disassembler** and **Zeus Disassembler**, $19.95 each, indispensible for machine language programmers.

And if you need software for your Coleco Adam, Softsync has **Personal Accountant**, a double-posting bookkeeping program for household and small business use, $34.94 for cassette; **Model Diet**, computerized meal planning based on nutritional requirements, $29.95; **Dancing Feats**, a one-man joystick band that lets even beginners play synthesized music instantly accompanied by synchronized color display, $29.95.

Softsync also offers many of these programs for Atari, Commodore 64, and IBM PC.



Inexpensive Word Processor

For your inexpensive printers, you need inexpensive word processing software. **Homeword**, from Sierra On-Line, comes with software, book, and tutorial audio cassette. Features include icons, or pictures of operations such as a file cabinet for filing, a page of print for editing, a printer for printing, and an unorganized page with an arrow pointing to an organized one for formatting. The specifications for this product seem similar to other word processors, but less usual features include a HELP key and the division of the screen into three sections. The upper section is largest and holds the working text. The lower right section shows a replica of the page as it will appear printed. The lower left lists available memory and disk space. The price is $49.95 for Apple II, II+, and IIe (soon available for Commodore 64 and Atari). Also included is a 30-day money-back guarantee.

Computers Talk to Each Other and the World

The **MITE/86** data communications program, from Mycroft Labs, turns computers into intelligent terminals to use with The Source, Compuserve, and the like, and to transfer files between 8- and 16-bit computers. You get full control over all communications parameters (parity, baud rate, etc.), automatic logon, three binary file protocols (XMODEM, CLINK/CROSSTALK, and Hayes), and everything else you'd expect of a good communications package for CP/M, for $195. The system is preconfigured for Rainbow, PC, Victor, and other CP/M-86 systems and is also available for various CP/M1-80 systems.

Low-Cost Printers

Alphacom offers a 40-column dot matrix printer, the **Alphacom 42**, that interfaces with most computers, for $99.95, complete with interface cable, or $79.95 without cable. It has upper and lower case and recognizes standard ASCII control codes. Alphacom also offers an 80-column, 100 cps printer, the **Alphacom 81**, for $214.95, including cable.

Basic BASIC

A good language needn't cost hundreds of dollars. The **Nevada BASIC** interpreter for CP/M systems, by Ellis Computing, is $39.95 for the diskette and documentation. The language includes IF . . . THEN . . . ELSE, BCD math (no rounding errors), 8- or 12-digit precision, single and multiline user functions, full matrix operations, and a full-screen text editor which can be accessed by pressing the RETURN key when a runtime error occurs.

Quiet, Fast Printer

Blue Chip Electronics has a 40 cps, 132-column, daisy wheel printer, called **Model BCD-4015**, with an optically controlled print head that automatically inserts paper and adjusts the left margin. The noise level is 57 decibels, which, they claim, makes it one of the quietest daisywheel printers available. It's bidirectional, logic seeking, and takes over 100 different printwheels. In addition to proportional printing, boldfacing, and underlining, a tractor feed is standard. There is an optional cut sheet feeder. Centronics 8-bit parallel interface is standard, and serial RS-232C, IBM PC, and IEEE-488 are optional. Warranty and after sale service are provided by General Electric Instrument and Communications Service Department, so maintenance and repair are available at 26 GE service locations in the US and Canada. The suggested retail price is $1,895.00.

Fast Z80 Forth

q4th for CP/M, from Quanta Corporation, is claimed to be the "fastest Forth for Z80 in the known universe." It's a superset of Forth-79, is an interpreter and also compiles to Z80 machine code, and has a runtime debug trace with stack display $95 includes one year update and newsletter subscription.

■■J

(Continued from page 82)

gence (AI).

Lately we have come to see some amalgamation of these techniques. Although the availability of cheap memory and higher speeds, along with a sophistication of inference rules, has greatly fortified the deductive approach, a judicious use of heuristic rules clearly is also necessary to allow the computer to escape from an infinite drudgery of trivial computation. In 1961 Marvin Minsky, one of the AI leaders, wrote: " . . . it seems clear that a program to solve real mathematical problems will have to combine the mathematical sophistication of Wang with the heuristic sophistication of Newell, Shaw, and Simon."

Summary

The present volumes concentrate on the deductive approach, perhaps justifiably since these techniques have so far yielded more substantial results. While Eliza-like programs are striking, they are not very sophisticated, either logically or heuristically.

In summary these are not books for the general hardware hacker but are of a rather narrow academic interest. They should be welcomed by their intended audience. ■■J

(Continued from page 96)

"I don't think, however, that you will ever get Unix wizards to follow these recommendations. I should have noted that I use Skjellum-like conventions when I'm writing micro-based applications. When I'm maintaining Unix sources I stick with the standard Unix conventions, which are pretty much Kernighan and Ritchie standard. Unix whizzes think that aligning curly braces is irrelevant, since "vi," the editor which 90% of them use, will always let you find the top or bottom match for any brace automatically. Also, and probably more important, Unix system guys always debug by staring at their CRTs, never from printouts (that's for COBOL programmers), and the goal is to reduce the number of lines of a function, so that as much of it as possible will fit on a screen. Seeing a *whole line* taken up by just an opening brace must drive them whacko, and some of them will even close blocks on the end of a line of code (yecch!)."

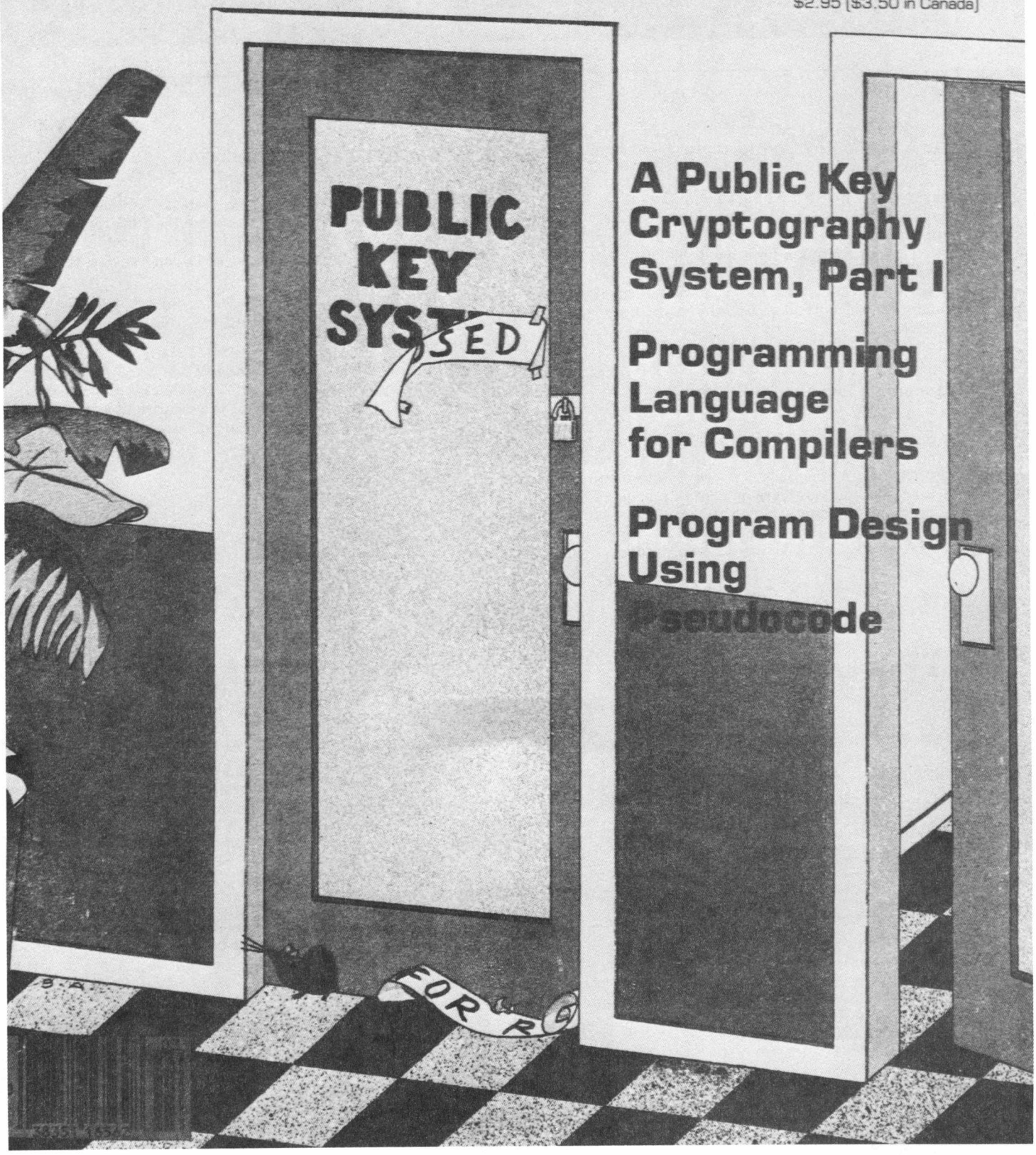
In this column, I have presented additions and corrections to the proposed layout standard, based on reader response. I have also presented the opposing point of view. Most of the letters received were positive, so it was difficult to include more dissenting remarks. I want to thank those who sent in their comments about the standard. ■■J

Dr. Dobb's Journal

For the Experienced in Microcomputing

#89 March 1984

$2.95 [$3.50 in Canada]



A Public Key Cryptography System, Part I

Programming Language for Compilers

Program Design Using Pseudocode

Dr. Dobb's Journal

For the Experienced in Microcomputing

March 1984
Volume 9, Issue 3

CONTENTS

ARTICLES

16 RSA: A Public Key Cryptography System, Part I
by C. E. Burton

This two-part article details implementation of the Rivest-Shamir-Adleman Public Key cryptography system on a microcomputer. This first installment includes a quick overview of the author's implementation language and a discussion of the mathematical algorithms used. Next month, Part II will describe the generation of the keys and the encryption/decryption system, and the use of digital signatures.

44 Introduction to PL/C: Programming Language for Compilers
by Morris Dovey

Because development of new and more powerful processors seems to be outstripping that of supporting software, the PL/C programming language was designed to fill the need for rapid, low-cost development of reliable programming language compilers. Included here are discussion of the language structure and syntax, and a listing of the compiler in PL/C and macro-assembly code.

70 Program Design Using Pseudocode
by Ken Takara

Program design can be tedious, even using some of the available design tools. This article uses the construction of a "shoot-em-up game" to examine the flexibility and convenience that using pseudo-code can provide in the design process.

78 More On Binary Magic Numbers
by Dale Wilson

Inspired by Edwin Freed's "Binary Magic Numbers" (*DDJ* No. 78, April 1983), this article provides a C language implementation of a number of the algorithms presented by Freed. This provides both useful functions and interesting examples of the use of C to implement them.

DEPARTMENTS

6 Editorial

7 Letters

9 Dr. Dobb's Clinic

Basically Precise; Transcendentally Precise; Stocking Up

12 CP/M Exchange

How fast is CP/M Plus?; CP/M 2.2 BIOS Function: SELDSK; CP/M V2.2 Application Note

84 Software Reviews

SXR PLUS: Sorted Cross Reference; Perfect Writer/Perfect Speller

88 Book Reviews

Cryptography: Proceedings of Workshop on Cryptography; IBM Data Files; Microprocessor Support Chips: Theory, Design, and Applications; Z80 Applications; Concepts for Distributed Systems Design; An Introduction to Numerical Methods with Pascal; Augmented Transition Networks

92 16-Bit Software Toolbox

The collected, sorted and tabulated responses to last September's floating-point benchmark

98 Of Interest

Publisher – Jane Nissen Laidley

Editorial
Editor – Reynold Wiggins
Managing Editor – Randy Sutherland
Contributing Editors –
Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg

Marketing
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Marketing Coordinator – Ed Gueble

Advertising
Advertising Director – Beatrice Blatteis
Advertising Sales –
Alice Hinton, Walter Andrzejewski

Circulation
Circulation Director – Terri Pond
Circulation Assistant – Frank Trujillo

Production
Art Director/Production Manager –
Barbara Ruzgerian
Production Coordinator –
Shelley Rae Doeden
Production Assistant – Alida Hinton
Typesetter – Jean Aring

Cover Illustration – Gregory A. Cerda

Copyright © 1984 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Donating Subscribers Contributing Subscriber: $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, recieves *five* one-year subscription).

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, Info World, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R. B. Sutton. **Lifetime Subscriber:** Michael S. Zick.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines and the current compensation schedule by sending a self-addressed, stamped envelope.

Foreign Distributors UK & Europe: Homecomputer Vertriebs HMBH 282, Flugelstr. 47,4000 Dusseldorf 1, West Germany; Euro Computer Shop, 182 Rue du Faubourg St. Denis, 75010, Paris, France; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – premiere, F-75014, Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** AXCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian Distributors)*

EDITORIAL

Several people found the time interval between the formal announcement of the telecommunications issue in our pages and the copy deadline to be a bit tight. Let that not be said about our *next* special issue. Be advised that our annual Forth issue is again planned for September, and the copy deadline will be May 14, 1984. We will provide more details next month, but interested parties may contact us at P. O. Box E, Menlo Park, CA 94026; (415) 323-3111.

We have a correction and an elaboration regarding last month's issue. On page 44, a telephone number was provided for testing one's VPC implementation. Unfortunately, the number given was the main number for Unir, not the VPC test line. The correct number to dial to test your VPC is (317) 842-6986.

The sidebar which accompanied the communications protocols article noted that a Monte Carlo simulation was being set up to estimate some of the accuracies for various error-checking schemes. Author Leslie Brooks has provided a Letter to the Editor this month which details the outcome of the simulation, and its effect on their initial predictions.

Below you will find the list of our current referees. As with other aspects of *DDJ*, we assume that the board will evolve over time. While we intend to list the referees that work on each issue, we will also publish a complete list periodically.

Our appreciation to those who have all along been informally providing technical advice and insights. Our special thanks to David E. Cortesi for his continuous and substantial support, and to Kim Harris for his willingness to look at so many Forth articles over the years.

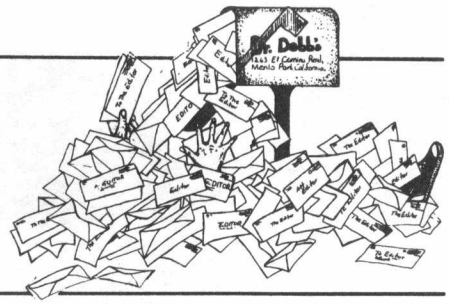
Reynold Wiggins

DDJ's Current Board of Referees

Dennis Allison, Computer Science Dept., Stanford University
Ian Ashdown, Professional Engineer
Joe Barnhart
S. M. Bellovin, AT&T Bell Laboratories
Robert Blum, Contributing Editor, *DDJ*
Patrick Burnstad, Lands End Computers, PATCA, SVCS, IEEE
Wayne Chin, Hewlett-Packard
David D. Clark, Department of Chemistry, Pennsylvania State University
David E. Cortesi, Contributing Editor, *DDJ*
Keith Coye, President Consulting Services Corporation and PicoNet
Mel Cruts, Diagnostic Engineer, Trilogy
Ray Duncan, Contributing Editor, *DDJ*
Michael T. Enright, Cal Omega Inc., ACM/SigGraph, IEEE
Z. Z. Fisher, Data Communications Consultants
Jim Fleming, Unir Corporation
Dr. Georges Grinstein, Computer Science Dept., Fitchburg State University
A. Gomez, Telecomp, Inc.
Kim Harris, Forthright Enterprises
J. E. Hendrix
William P. Hogan, Owner, William Hogan Associates,
David W. Hopper, Senes Consultants Limited
J. C. Johnston, Graduate Student Cleveland State University
Michael P. Kelly, Design Software
John P. Keyes, Student Syracuse University
David Kirkland, Baker & Botts
Richard G. Larson, Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago
Ben A. Laws, Assistant Professor, Computer Science, North Texas State University
Mohamed el Lozy, M.D., Ph.D., Harvard School of Public Health
Patrick Lynch, Tandem Computers Inc. of Cupertino
Clay Phipps, ACM
William Ragsdale, President, Forth Interest Group
Donald L. Rastede
Richard A. Relph
John A. Rogers, Fortune Systems
Douglas W. Rosenberg, Optimal Software
David Ross, Professor, University of Iowa
Darryl E. Rubin, ROLM Corporation
Joseph Sharp, Micro Science Associates
Henri Jonathan Socha, SochaLogical Research, IEEE, ACM
John K. Taber, IBM
Scott D. Thomas, Informatics General Corp.
Allen Tigert, Kontron Medical Electronics
Bob Wheeler, Custom Hardware
Charles Wilde, Aton International Inc., IFEE, ACM

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94026. Second class postage paid at Menlo Park, California 94026 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94026 · 415/323-3111 **ISSN 0278-6508**

LETTERS



The editorial response card is a great way to talk to us, but don't forget that Dr. Dobb's Journal *also welcomes letters to the editor as a forum for ideas, innovations, irascibility and even idiosyncrasies. Some letters may be edited for clarity and brevity. The Doctor likes hearing from you – keep on writing.*

Method:	XOR	Sum	Sum w/carry
Random Errors:	1 in 136	1 in 187	1 in 221
Clustered Errors:	1 in 417	1 in 381	1 in 360

Table
Probability of Missing an Error

Accuracy Update

Dear Doctor:

In our recent article on communications protocols (February 1984, *DDJ* No. 88), my friend John Rasp and I were discussing the relative merits of various methods of error detection. As we said there, the mathematics becomes *very* tricky if the changed bits are clustered together rather than randomly distributed. In order to get accurate results for the clustered errors, and to check our results for random errors, we finally ran a Monte Carlo simulation on the university's Cyber 760 mainframe. The results were very enlightening; although he couldn't prove it John had not expected the two cases to differ, but in fact they did. It turns out (as you can see by looking at the table top right) that all of the common methods of error detection are significantly better at catching errors if the changed bits are clustered together. In fact, the XOR method of calculating a checksum is the *worst* method of calculating the checksum if the errors are random, but the *best* method if the errors are clustered.

These results should be accurate to 3 digits. The simulation ran 40,000 iterations, generating a block of 128 bytes of random data each time, then randomly choosing the first bit to be clobbered, then randomly choosing the number of bits to be clobbered, then choosing the values of the clobbered bits. The number of bits to be hit was clustered around 10, in the usual bell curve. The simulation was written in Fortran, and required 15 minutes of CPU time to run (on a 10 megaflop machine)!

I hope this clears up any lingering questions.

Sincerely,
Leslie Brooks
Computing Center
The Florida State University
Tallahassee, FL 32306

Fast Conic Curves

Dear *DDJ*:

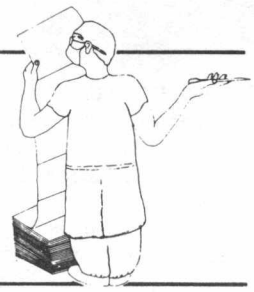
A general algorithm for drawing any conic curve at any orientation uses a slightly different viewpoint than the one detailed by Michael Enright (*DDJ* No. 86, December 1983, pp.19-20), but still yields all the benefits of his method. Assuming the cursor starts on the curve, the best direction of the four possible directions is chosen by evaluating the potential distance that each would be from the actual curve and taking the one that gives the smallest. This distance is found by using a form of the gradient which for conic equations is linear, i.e., using addition not multiplication. By updating a table of possible changes again using only addition, the gradient at each point can be maintained. The algorithm continues in this way until it nears the endpoint. The algorithm draws the "best possible" curve – the one closest to the actual curve.

I developed this method three years ago for drawing circles and circular arcs. In the degenerate case it yields the standard method for drawing lines. With the growing speed and capacities of microprocessors, I believe that algorithms that "look before they leap" will be increasingly useful.

Jim Hatton
3715 Summit Drive
Mt. Shasta, CA 96067

■■J

DR. DOBB'S CLINIC



by D.E. Cortesi, Resident Intern

Basically Precise

That describes the material sent us by Allan Behler, printed in the November Clinic. Allan had discovered that, in Microsoft BASIC-80, a constant assigned to a double-precision variable wasn't stored as expected:

```
DEFDBL A
A=134.12
PRINT A
134.1199951171875
```

The display could be made correct with PRINT USING, but repeated use of such values could accumulate significant errors. With a good deal of difficulty he worked his way to a solution using a rounding function to convert from a single-precision value to a double-precision one. Here is a corrected, improved version of the rounding function shown here in November:

```
10 DEF FNRD#(X#)=
    INT(X# *100+0.5)/100#
20 DEFDBL A
30 A=134.12 : PRINT A
40 A=FNRD#(A) : PRINT A
RUN
134.1199951171875
134.12
```

Three things are needed to make the function work as expected. First, there are two expressions in FNRD# (one is the argument of INT, the other is the division), and at least one of the elements of each expression must be double precision. Why? Read the following rule from the *BASIC-80 Reference Manual* (Microsoft, 1979, page 1-8):

> "During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand."

In other words, if the divisor isn't "100#" indicating a double-precision constant, the result of the division will be single precision. The single-precision result will be stretched to double to satisfy the "#" in the name of the function, and the result will be exactly the same as assigning 134.12 to A in the first place.

Second, the function name has to indicate that it returns a double-precision result. If it doesn't – if its name is simply FNRD, for example – the double-precision result of the expression will be truncated to single precision to match the implicit type of the function. Then *that* result will be stretched to double precision for the assignment to A, and we are back to square one.

All these problems are the effects of another rule (*Ibid*, page 1-9):

> "If a double-precision variable is assigned a single-precision value, only the first seven digits, rounded, of the converted number will be valid. [*Note: it's six digits in the IBM manual.*] The absolute value of the difference between the printed [*sic*] double-precision number and the original single-precision value will be less than 6.3E-8 times the original . . . ."

It isn't clear to us why, after a constant has been converted correctly to a single-precision float value, copying it to a double-precision float value should change its low-order bits (if that's what happens). However, the effect is there, it's documented, and it's dangerous.

Finally, the function can still fail if it doesn't indicate that it requires a double-precision argument. If its dummy argument is named X rather than X#, and if the value of the argument has more than six significant digits, the result is wrong. The (effectively double-precision) argument gets truncated to single precision when assigned to the dummy variable, and data is lost.

Well, Allan's record of his explorations provoked some sharp replies. Let's read some of them. Charles Marshall says, "If the example is rewritten to use a double-precision constant, the correct answer is printed:

```
A=134.12# : PRINT A
134.12
```

"Mr. Behler continues to use mixed-mode arithmetic in the three program segments, which will tend to exaggerate the problem he is trying to solve. In addition his use of 0.51 as a rounding constant is invalid, resulting in, for example, the number 1.0049 being 'rounded' to 1.01."

Rounding is another whole topic. We think Marshall is right, but would anyone care to guess why Allan might have (very deliberately) used 0.51 instead?

Meanwhile, Joseph McDermott writes:

> "People have been stumbling over this problem since 1977. I found the correct solution on my TRS-80 in 1978, and it still applies to the IBM PC. Why does not Microsoft explain it is their manuals and save everybody so much grief? Numeric precision problems disappear if a few simple rules are followed."

McDermott's Rule #1 is: *All values entering a calculation must be double precision.* You must be consistent from beginning to end of your program, *no exceptions*, he says. This extends to things like input where, if you read numeric input as a string and then convert it, you must remember to append the magic pound-sign:

```
INPUT NUM$
NUM# = VAL( NUM$ + "#" )
```

"BASIC reverts to single precision at the slightest excuse," he says. "A double-precision variable on the left of the assignment statement is *not* sufficient to enforce double-precision calculations."

Joseph Sabin wrote in with what we think is a false lead: "Whenever you change a single-precision number to a double-precision number, you must use the double-precision exchange function:

```
A# = CDBL(123.12)
```

or you will get inaccurate results."

A good point, we thought, until we tried it:

```
A# = CDBL(134.12)
PRINT A#
134.1199951171875
```

Oops. Sorry, Joseph, we can't find any difference between using CDBL and not. It is definitely not a replacement for the use of a double-precision marker on the constant itself – "134.12#" and "CDBL(134.12)" do not appear to be the same thing. In fact, the documentation for CDBL reveals exactly that. Take this example, exactly as shown in the IBM BASIC manual (first edition, page 4-31):

```
10 A = 454.67
20 PRINT A; CDBL(A)
RUN
454.67  454.6699829101563
```

That's Behler's original problem; CDBL does *not* do what FNRD# does. What FNRD# does is probably not useful when working with constants; the trailing pound sign does it better. Its only use would be in the rare cases when you have a computed value of money in a single-precision variable and want to assign it to a double-precision variable. In that specific case it does a better job that CDBL.

Transcendentally Precise

Richard Falk writes asking for help in finding, "either a machine code source listing or a fast algorithm for trig functions on the 6502. I've coded the power series for sin, cos, tan, arcsin, and arctan, but these require so many terms for sufficient accuracy (12-13 decimal digits) that they run too slowly." He notes that the best sin function he's come up with takes 500 ms to execute, compared to 340 ms for the BASIC sin function on the same machine. Would anyone care to recommend a good reference for this? We suppose that Falk would like a cookbook approach, rather that a course in numerical analysis.

Stocking Up

Dick Mesirov is the kind of reader we really appreciate: He sent us some original, unsolicited input – a "sponse," not a re-sponse. (Go thou and do likewise!) Here's how he tells it:

"I am a market maker on the Philadelphia Stock Exchange Options floor. If you've seen TV shots of those guys screaming, hollering, and waving their arms in the commodity pits, well, that's what I do. I store data on the stocks I trade: opening price, high, low, close, and the direction of last trade (up or down). This data is entered and used daily, and I store it by month. That is, all of October's entries are stored in the file named *xxx*OCT83, where *xxx* is the three-letter symbol for the stock.

"When I run the program each evening, after entering the day's data, I recall several months' data in addition to the current month's. The number of months required varies depending on what I am trying to do. Originally I recalled each month by name; then I came up with the routine listed here. It recalls and saves one month's data a day at a time, then automatically steps down by one month and recalls the prior month's data. After getting January's data it also steps down one year.

"I've never seen anything like it in any of the books I have nor in any of the commercial programs I've looked at. The same approach could be used by day or I guess for any series that can be listed as a string."

Dick's routine appears in Listing One (below). The original was cleanly structured using IFs and GOTOs; we took the liberty of recoding it to use nested FOR and WHILE loops to emphasize its shape. The only problem we can see is not in the program but in CP/M. There is a definite limit to the number of files you can store on a disk. Typically there are only 64 or 128 entries in the disk directory. When Mesirov accumulates a year's data on ten stocks he can look forward to getting a file error from filling up the directory. Within that limit, this looks like a useful technique. ■■J

Dr. Dobb's Clinic (Text begins on page 9)

```
9000 REM
9010 REM   SUBROUTINE TO READ STOCK HISTORY FILES
9020 REM history files are named SSSMMMYY, where
9030 REM   SSS=stock-id, MMM=month, YY=year
9040 REM returns N=number of entries read
9050 REM
9060 IM$="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
9070 INPUT "NAME OF STOCK";ST$
9080 INPUT "CURRENT MONTH";MO$
9090 B=INSTR(IM$,MO$)
9100 INPUT "CURRENT YEAR";YC$ : YC=VAL(YC$)
9110 INPUT "HOW MANY MONTHS' DATA NEEDED";NM
9120 N=0
9130 FOR JM=1 TO NM
9140    REM read one file's (month's) data
9150    OPEN "I",#1,ST$+MO$+YC$
9160    WHILE NOT EOF(1) : N = N+1
9170       REM here READ a stock entry into various
9180       REM ..arrays subscripted by N
9190    WEND : CLOSE
9200    REM get the prior month and year
9210    B=B-3
9220    IF B<1 THEN B=B+36 : YC=YC-1 : YC$=RIGHT$(STR$(YC),2)
9230    MO$=MID$(IM$,B,3)
9240 NEXT JM
9250 RETURN
```

End Listing

CP/M EXCHANGE

by Robert Blum

Last month I ran a preliminary application note for CP/M Plus that reportedly would optimize the access time to any disk file – especially one that was small enough to fit into buffer memory and was accessed more than once within the same program. After installing the patches on my system, there appeared to be a significant increase in overall system performance, although at that time I had nothing more concrete to go on than an impression.

How Fast Is It?

To find out how much of a speed advantage (if any) was gained, I wrote a small benchmark program to be used in comparing the performance between an off-the-shelf distribution system and a modified one. The benchmark program I wrote had five distinct phases. The first four phases performed preliminary maintenance functions: make four files; write 16K of garbage data to each file; close the four files; and open the same four files that were just created.

The fifth and final phase sequentially read and reread the four files four times. If the changes were to be a success, this final phase of the program would produce the most dramatic evidence.

After completing the benchmark program, I copied the unaltered distribution files of my CP/M Plus system into a separate user area for use in link-editing the standard or unaltered CP/M system used in my benchmarks. My last step was to apply the patches outlined in the application note to one copy of my CP/M Plus system.

As shown in Table I (at right), four benchmarks were run, each with different combinations of features, etc. The first benchmark was run with two CP/M Plus systems that were completely stripped down and (as closely as I could make them) equal in performance to their V2.2 predecessor. As you can see, the runtime difference between the distribution and the modified system is small enough to be inconsequential.

The second and third benchmarks produced a healthy runtime reduction over the first one, but little difference was found between the two of them. This lack of improvement was a total surprise to me since I had expected the LRU buffering improvements to be more significant no matter what other options had been selected. But, as it appears, the LRU buffering logic is directly, or at least very closely, tied to the logic used for directory hashing.

Benchmark number four plainly points out how much of an improvement a plain vanilla CP/M Plus system can make in runtimes (not to mention the further reductions that can be experienced by tuning the system a little).

CP/M 2.2 BIOS Function: SELDSK

Billy Smith of Kentfield, California, writes:

"Here is a special treat for CP/M hackers. This tidbit just turned up as the root of a tricky little bug. I was implementing a public domain program called FILE.ASM at the time. Its function is to display all files on all drives and user areas matching the ambiguous file reference given in the command line. On my system it was stopping after completing drive A as if I had just done a cold boot and hadn't referenced any other drives yet.

"Debugging revealed that there is a slight difference in the version 2.2 CP/M BIOS function SELDSK from all earlier versions. There is an additional sentence in the V2.2 manual, under SELDSK, that explains a little further: 'The least significant bit of register E is zero if this is the first occurrence of the drive select since the last cold or warm start.' There was no reference to register E having to be preset to any special value in earlier versions of the CP/M manual.

"My BIOS (Morrow) takes advantage of this information and does not do a formal selection of a drive if the bit is nonzero. Instead it simply returns a value from a local variable that is assumed properly set during the first disk select. I suppose CP/M keeps this bit correct when calling SELDSK, but any program that does direct BIOS calls has the responsibility of managing this bit.

"One problem with this logic is that an application program has no idea if a drive has already been referenced or not and therefore must always set the bit to zero for its first reference even though this may be redundant to the BIOS. Since this public domain program had random nonzero data in register E, my BIOS was returning a bad value from SELDSK. In fact the bad value happened to be zero, which the program interpreted to mean non-existent drive and, of course, ended its operation.

"My simple fix was to clear register E before the call to SELDSK. Every call to SELDSK is now treated as though it were the first disk select of that drive and a proper value is returned."

Application Note 2

Michael Carter of Garran, Australia, wrote several months ago to share a patch he had developed for reversing the meaning of the BACKSPACE and DELETE keys in CP/M V2.2. Michael's note prompted me to dig through my files to see if DR had officially released anything

Run	Distribution	Modified
#1	76 seconds	74 seconds
#2	64 seconds	59 seconds
#3	62 seconds	61 seconds
#4	37 seconds	25 seconds

Legend

#1 No features enabled, 1 Directory buffer and 1 Data buffer
#2 Directory Hashing enabled, 1 Directory buffer and 1 Data buffer
#3 No features enabled, 23 Directory buffers and 255 Data buffers
#4 Directory Hashing enabled, 23 Directory buffers and 255 Data buffers

Table I.

Distribution vs. Modified CP/M Plus

on this subject. Sure enough, in February of 1982 an application note was published that thoroughly covers the subject.

CP/M® V2.2 Application Note 02, 2/20/82: Reversing the BACKSPACE and RUBOUT Key Functions and Making RUBOUT Identical to BACKSPACE

Copyright © 1982 by Digital Research. CP/M is a registered trademark of Digital Research. DDT and SID are trademarks of Digital Research. Compiled November 1982. Reprinted with permission of Digital Research. All information here is proprietary to Digital Research.

Applicable products and version numbers:
CP/M® V2.1 and V2.2

Program: BDOS

In the following code segment procedures, addresses given are hexadecimal offsets from the base of the CP/M system. The CCP is usually located at 980H but can be located at A00H if a two-sector boot is used.

You can assemble the patch for your size memory system. The cpmbase equals the BDOS entry point address at locations 6 and 7 in the base page of memory minus 806H. You must change this entry point address when you load DDT™ or SID™. Under DDT or SID, follow the jump at location 5 until an address is found with a least significant digit of 6. In the following example, the cpmbase would be E506H-806H or DD00H.

```
0005    JMP     CD00
CD00    JMP     D3A4
D3A4    XTHL
D3A5    SHLD    E452
D3A8    XTHL
D3A9    JMP     E506
```

Procedure to reverse the BACKSPACE and RUBOUT key functions:

Patch into the SYSGEN or MOVCPM image exactly as you would patch in a new version of your BIOS, using the DDT i command followed by the DDT r command. You can use the same offset as your custom BIOS and install the code found in Listing One (page 14).

Patch into the SYSGEN or MOVCPM image exactly as you would patch in a new version of your BIOS, using the DDT i command followed by the DDT r command. Use the same offset as your custom BIOS and install the code in Listing Three (page 14).

Or, you can install the above procedure directly into MOVCPM if you have MOVCPM.COM on your system disk. The patch is installed automatically in any size system that you build using MOVCPM. Make a back-up copy of MOVCPM.COM before using DDT to make the following changes:

```
A>ddt movcpm.com
DDT VERS 2.2
NEXT PC
2700 0100
-1141b
141B    MOV     A,B
141C    ORA     A
141D    JZ      09EF
1420    MOV     A,M
1421    DCR     B
. . .
-a141b
141B mvi a,8
141D jmp a07
1420 .
-g0
A> save 38 movcpml.com
```

Use the new program MOVCPM1 .COM in place of MOVCPM.COM. The RUBOUT and BACKSPACE key functions are identical in any CP/M system generated with MOVCPM1.COM.

Licensed users are granted the right to include these modifications in CP/M V2.2 software.

Or, you can install the above procedure directly into MOVCPM if you have MOVCPM.COM on your system disk. The patch is applied automatically to any size system that you build using MOVCPM. Make a back-up copy of MOVCPM.COM before using DDT to make the following changes:

```
A>ddt movcpm.com
DDT VERS 2.2
NEXT PC
2700 0100
-11402
1402    CPI     08
1404    JNZ     0A16
1407    MOV     A,B
1408    ORA     A
1409    JZ      09EF
140C    DCR     B
140D    LDA     0B0C
1410    STA     0B0A
1413    JMP     0A70
1416    CPI     7F
1418    JNZ     0A26
-s1403
1403 08 7f
1404 C2 .
-s1417
1417 7f 8
1418 C2 .
-g0
A>save 38 movcpml.com
```

Use the new program MOVCPM1 .COM in place of MOVCPM.COM. The BACKSPACE and RUBOUT key functions are reversed for any CP/M system generated with MOVCPM1.COM.

Procedure to make RUBOUT identical to BACKSPACE:

Before you install this patch, the code at cpmbase + 0A1Bh should read as shown in Listing Two (below). ■■J

CP/M Exchange Listing (Text begins on page 12)

Listing One

```
cpmbase equ     ?       ;subtrack 806h from address
                         at location 6
        org     cpmbase + 0A02h
        cpi     7fh     ;was cpi 08h
        org     cpmbase + 0A16h
        cpi     08h     ;was cpi 7fh
```

End Listing One

Listing Two

```
mov     a,b
ora     a
jz      cpmbase + 09EFh
mov     a,m
dcr     b
dcx     h
jmp     cpmbase + 0AA9h
```

End Listing Two

Listing Three

```
cpmbase equ     ?
;
;
        org     cpmbase + 0A1Bh
;
        mvi     a,8h
        jmp     cpmbase +0A07h
        end
```

End Listing

RSA: A Public Key Cryptography System, Part I

"In this two-part series of articles, we will discuss the Rivest-Shamir-Adleman [RSA] PKS and show how to implement it on a microcomputer."

Cryptography has been in existence since the beginning of written history. Most of these ciphers and codes were developed for military applications. One of the more well-known systems, designed by Julius Caesar, is a simple letter transposition cipher that is easily broken today. Since that time, cryptography has matured; some current systems are estimated to be unbreakable, even using state-of-the-art technology. Some claim that cryptographers would have to spend millions of years using today's fastest computers to break some of these ciphers.

Key Systems

In the late 1970s, IBM introduced a single key system that was later made into a standard by the National Bureau of Standards. The Data Encryption Standard (DES) is a 56-bit key system that has been committed to silicon by several semiconductor vendors. The cipher is fast and easy to integrate into LSI circuits. During and since its standardization, controversy has raged over the security of the DES. The primary concern is over the short length of the key. Many also speculate that since the government, which has overriding national security concerns, standardized the cipher, they may have a means of breaking it. However, many groups use the DES, including financial institutions, local network manufacturers, and others.

The Public Key System (PKS) was first proposed by W. Diffie and M. E. Hellman in 1976.[1] Their paper described a dual key system with keys generated from large prime numbers (100+ decimal digits). In 1979, Hellman published an article in *Scientific American* describing the mathematics of the PKS.[2] He also offered a monograph that described the system in greater detail, but within a year of the monograph offering, it was withdrawn from public distribution. Speculation was that the government (NSA and DoD) had forced its withdrawal because it jeopardized the national security. Unlike the DES cipher, the PKS has not been committed to silicon because it is much more complex and requires considerably more mathematics and computation time, as we will soon see. Currently, investigators continue ongoing research efforts to generate efficient integrated circuit implementations of the PKS.

In 1978, R. L. Rivest, A. Shamir, and L. Adleman showed how the PKS could be implemented and proposed a means of providing digital signatures to the messages.[3] The addition of a signature to a message allows a sender to sign his message so that a receiver can be sure that the message originated with the sender. The February 1983 issue of IEEE *Computer* had a series of articles discussing ways to compromise digital signatures and how they could be made more secure.[4, 5, 6, 7] Some of the articles examined several different ciphers, their basic implementations, and their limitations.

In this two-part series of articles, we will discuss the Rivest-Shamir-Adleman (RSA) PKS and show how to implement it on a microcomputer. The first part will discuss RATFOR (the implementation language) and the mathematical core of the RSA system, including modulo arithmetic, multiple-precision arithmetic, and "Russian Peasant" exponentiation. In the second part, we will describe the generation of the keys (public and private) and the encryption/decryption system; we will also take a brief look at digital signatures.

RATFOR

RATFOR (RATional FORtran) is a preprocessor to Fortran; i.e., the output of the RATFOR precompiler is Fortran. The language is described in the book *Software Tools*.[8] RATFOR adds several structured constructs to Fortran while allowing use of standard Fortran statements, in-line comments, free-form input, multiple statements in a line, definitions, inclusion of other files, logical conditions similar to BASIC or C, a standard library, and so on. In a lot of ways, it is similar to the C programming language, and those familiar with C should have little trouble transporting RATFOR source to C source. For that matter, those familiar with Pascal should be able to transport the software easily.

Those who want to use RATFOR can obtain a public domain version from the CP/M Users Group (CPMUG), operated by LifeBoat Associates, or from one of the RCPM bulletin boards that has it available for downloading. Unfortunately, a library is unavailable with the public domain version; however, using the *Software Tools* text, you can generate your own library of functions or simply pull out the appropriate functions from the precompiler.

Now let's review some of the rudiments of the language. The structured constructs that RATFOR adds to Fortran include:

```
repeat
    {
    .
    .
    statement n
    .
    }
until (condition)

while (condition)
    {
    .
    .
    statement n
    .
    .
    }

for (init. statement; condition; loop
        statement)
    {
    .
    .
    statement n
    .
    .
    }

do n=j,k    # Note the lack of a ter-
            # minating label number
    {
    .
    .
    statement n
```

by C. E. Burton

C. E. Burton, 1720 S. Deframe Court, Denver, CO 80228.

Copyright © 1983 by C. E. Burton. All rights reserved. Permission is granted for personal, noncommercial use only.

```
        .
        .
        }
    if (condition)
        {
        .
        .
        statement x
        .
        .
        }
    else if (condition)
        {
        .
        .
        statement y
        .
        .
        }
    else
        {
        .
        .
        statement z
        .
        .
        }
```

RATFOR uses # as an in-line comment indicator and ignores anything after its occurrence to the end of the line. The { and } symbols are used as Begin and End designators, respectively. Within the iterative constructs, you can use the special words *break* and *next*; *break* causes an exit from the innermost loop, while *next* causes the innermost loop to continue execution at its "condition" statement. Because of the form of the "if" statement that RATFOR uses, you cannot use a Fortran arithmetic IF within RATFOR. For example,

```
if (condition) less-label,equal-label,
    greater-label
```

is a "no-no."

The "condition" statements can contain any of the following logical operators:

==	Equals
!=	Not Equal
<	Less Than
<=	Less Than or Equal
>	Greater Than
>=	Greater Than or Equal
!	Not
&	And
|	Or

The last three logical operators, !, &, and |, can also be used as logical bitwise operators on logical or integer variables.

Two other special words that are available are *define* and *include*. *define* allows you to make substitutions of strings within the source file. Beginning where it occurs within the source, it substitutes the right string for the left string throughout the text. An example follows:

```
define(Yes,1)   # substitute "1" for
                # "YES" before processing
```

include allows you to include another file at the point that the statement occurs.

```
if ( ( (variable1 == variable2) & (variable3 != variable4) ) |
     (variable5 < variable6) )

call subroutine(variable1,variable2,variable3,variable4,
                variable5,variable6)

longline=variable1*(cos(variable2)/tan(variable3)+variable4*
         variable5)-variable6
```

Figure 1.

For example:

```
include RATDEF    # reads and com-
                  # piles the file RATDEF.RAT
```

Multiple statements per line are handled using semicolons, as shown below:

```
statement 1; statement 2; statement 3
```

Statements can also be extended across multiple lines. The "condition" statements are automatically extended; statements with comma separators are extended if the comma ends the line; but other statements must use an underscore (_) to extend across a line boundary. Figure 1 (above) illustrates these three extensions. The precompiler limits the line to 72 characters, generates a continuation character in column 6 of the next physical line, and continues the source in this fashion until the logical line ends. To get an idea of how RATFOR handles these items, consult Listing One (page 22) and Listing Two (page 24) for examples of the RATFOR source and Listing Three (page 24) for the Fortran source generated by the precompiler.

Modulo Arithmetic

At the heart of a PKS cipher is the ability to perform multiple-precision, nonnegative, integer arithmetic. As was implied earlier, the PKS cipher requires the use of addition, subtraction, multiplication, division, and exponentiation of large numbers (100 to 250 decimal digits). Most computers, including microcomputers, are limited to arithmetic on numbers having 3 to 10 decimal digits (usually in binary form). To compound the problem, performing the RSA algorithms requires the use of modulo arithmetic. In some respects, modulo arithmetic is a lifesaver in that it limits the size of the numbers that are generated; however, it also complicates things because additional processing must be done on the results of arithmetic operations.

While modulo arithmetic is similar to the arithmetic that we learned in school, it has some differences. Thus, we should look at a few of its properties. A "modulus" is essentially the base of the number system that is being employed, and the mod operator leaves the "residue" of numbers. We could define the operation (A mod B) for nonnegative integers (in which we are currently interested) as:

$$A \bmod B = A - (B * \text{int}(A / B)) = \text{remainder}(A / B)$$

Therefore, a number system with a base of 10 (decimal) can only contain the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The following example uses hex (base 16):

$$75 \bmod 16 = 75 - 16 * \text{int}(75/16) = 75 - 16 * 4 = 75 - 64 = 11 < 16$$

Another interesting property of modulo arithmetic is:

$$(A \mathbin{@} B) \bmod C = ((A \bmod C) \mathbin{@} (B \bmod C)) \bmod C$$

where @ can represent +, –, or *. Exponentiation is similar:

$$(A ** B) \bmod C = ((A \bmod C) ** B) \bmod C$$

These operations are also commutative; i.e., the order of performance is not important, as illustrated in Figure 2a (at right). Figure 2b (below) shows a couple of examples to make sure we have it.

Now you can understand why using modulo arithmetic in raising large numbers to large powers can be a lifesaver. However, it does add considerably more operations (one division to get the remainder after each operation).

Multiple-Precision Arithmetic

In this section, we will cover the basic operations of addition, subtraction, multiplication, and division of nonnegative integers. I should point out that there are a few programs capable of performing multiple-precision arithmetic on very large numbers, e.g., MUMATH. These programs undoubtedly use similar algorithms to perform their operations, and I am sure that they could be put to use effectively in implementing the RSA algorithm. Our current objective, however, is to show how to do it.

The general opinion is that D.E. Knuth has written the "bible" of computer algorithms, and his work is the basic source of most of what I will subsequently pre-

sent in this set of articles. Many of the extensions to what he presents in the text of his books can be found in the problems at the end of each section and in the subsequent answers at the end of each book.[9] Another paper dealing with multiple-precision arithmetic that might be helpful here was written by M. Zimmerman.[10] My implementation of the multiple-precision arithmetic (MPARITH.RAT) can be found in Listing Four (page 26).

Looking at the basic arithmetic operations, we can say that:

1. Addition of two N-digit numbers produces a result that has at most (N+1) digits.
2. Subtraction of two N-digit numbers produces a result that has at most N digits.
3. Multiplication of an N-digit number by an M-digit number produces a result that has at most (N+M) digits.
4. Division of an (N+M)-digit number by an N-digit number produces a quotient of at most (M+1) digits and a remainder of at most N digits.

These statements hold true no matter what radix (number base) system is used. In my particular case, I have used a radix of 128, where each byte holds a number between 0 and 127, because the Fortran that I used (Microsoft F80) does two's-complement arithmetic on byte variables. A *define* statement near the beginning of the routines to be presented in Part II sets BYTEMODULUS to 128. The MPARITH routines shown in Listing Four, however, are not limited to this value. They allow any byte modulus (radix) between 2 and 128, and the value is passed to the subroutine by the parameter MODULO.

Knuth, in his set of books dealing with computer programming, presents both algorithms and the "MIX" assembly language programs that implement the algorithms. I have implemented a RATFOR (Fortran) version of the algorithms. No doubt, an assembly version of the algorithms written in Microsoft's M80 would run considerably faster, but I was more interested in transportability and ease of understanding than in speed. (A basic premise of programming is to get it working first then to modify things to optimize performance only if it is necessary. Another premise is that if it ain't broke, don't fix it!)

I will present Knuth's algorithms here without going into detail on my routines.* Since RATFOR allows in-line comments, the code should include most of the required documentation, especially when it is compared to the original algorithms. A transcription of these algorithms follows with some minor rewording. These algorithms are presented for those of you who do not have access to Knuth's second volume.

Addition Algorithm

ALGORITHM A (addition of nonnegative integers). Given nonnegative N-place integers (u1, u2, . . . , un) radix b and (v1, v2, . . . , vn) radix b, this algorithm forms their sum (w0, w1, w2, . . . , wn) radix b. (Here w0 is the carry, and it will always be 0 or 1.)

A1. [Initialize] Set j=n, k=0. (The variable j will run through the various digit positions, and the variable k keeps track of carries at each step.)

*A2. [Add digits] Set wj=(uj + vj + k) mod b and k=floor((uj + vj + k) / b). (In other words, k is set to 1 or 0, depending on whether a carry occurs or not, i.e., whether uj + vj + k >= b or not. At most one carry is possible during the two additions, since we always have uj + vj + k <= 2 * (b - 1) + 1 < 2 * b, by induction on the computation.)*

A3. [Loop on j] j=j-1. If j > 0, then go to A2; else set w0=k and terminate.

Note: floor(x) is the greatest integer less than or equal to x, also known as int(x).

Subtraction Algorithm

ALGORITHM S (subtraction of nonnegative integers). Given nonnegative N-place integers (u1, u2, . . . , un) radix b >= (v1, v2, . . . , vn) radix b, this algorithm forms their nonnegative difference (w1, w2, . . . , wn) radix b.

S1. [Initialize] Set j=n, k=0.

*S2. [Subtract digits] Set wj=(uj - vj + k) mod b and k=floor((uj - vj + k) / b). (In other words, k is set to -1 or 0, depending on whether a borrow occurs or not, i.e., whether uj - vj + k < 0 or not. In the calculation of wj, note that we must have -b <= uj - vj + k <= b or 0 <= uj - vj + k + b <= 2 * b.)*

S3. [Loop on j] j=j-1. If j > 0, then go to S2; else terminate. (When the algorithm terminates, k should be 0; the condition k=-1 will occur if and only if V > U, and this is contrary to the assumptions.)

Multiplication Algorithm

*ALGORITHM M (multiplication of nonnegative integers). Given nonnegative integers (u1, u2, . . . , un) radix b and (v1, v2, . . . , vm) radix b, this algorithm forms the product (w1, w2, . . . , wl) radix b, l=m+n. (The conventional pencil-and-paper method is based on forming the partial products (u1, u2, . . . , un) * vj first, for 1 <= j <= m, and then adding*

Donald E. Knuth, The Art of Computer Programming, Vol. 2, ©1981, Addison-Wesley, Reading, Massachusetts. Pgs. 250, 252, 253, 254, 257, 258 and 442 (to include Algorithms "A", "S", "M", "D", "A"). Reprinted with permission.

```
(A @ B) mod C = (B @ A) mod C

(A ** (B*C) ) mod D = ( (A ** B) ** C) mod D
                    = ( ( (A ** B) mod D) ** C) mod D
                          — or —
                    = ( (A ** C) ** B) mod D
                    = ( ( (A ** C) mod D) ** B) mod D
```

Figure 2a.

```
(5 * (6+2) - 4) mod 7 = 36 mod 7 = 1
                      = (5 * (8 mod 7) - 4) mod 7 = (5 - 4) mod 7 = 1

(3 ** (4*2) ) mod 7 = 6561 mod 7 = 2
                    = ( ( (3 ** 4) mod 7) ** 2) mod 7
                    = ( (81 mod 7) ** 2) mod 7 = (4 ** 2) mod 7
                          — or —
                    = 16 mod 7 = 2
                    = ( ( (3 ** 2) mod 7) ** 4) mod 7
                    = ( (9 mod 7) ** 4) mod 7 = (2 ** 4) mod 7
                    = 16 mod 7 = 2
```

Figure 2b.

these products together with appropriate scale factors; in a computer it is best to do the addition concurrently with the multiplication, per this algorithm.)

*M1. [Initialize] Set wj=0, m+1 <= j <= m+n. Set j=m. (If wj above were not set to zero in this step, it turns out that the steps below would set W = U * V + (wi, . . . , wj), i=m+1 and j=m+n. This more general operation is sometimes useful.)*

M2. [Zero multiplier?] If vj = 0, set wj=0 and go to Step M6. (This test saves a good deal of time if there is a reasonable chance that vj is zero, but otherwise it may be omitted without affecting the validity of the algorithm.)

M3. [Initialize i] Set i=n and k=0.

*M4. [Multiply and add] Set t = ui * vj + (wr + k), r=i+j; then set wr = (t mod b) and k = floor(t/b). (Here the carry k will always be in the range 0 <= k < b.)*

M5. [Loop on i] i=i-i. Now if i > 0, go back to Step M4; otherwise set wj=k.

M6. [Loop on j] j=j-1. Now if j > 0, go back to Step M2; otherwise the algorithm terminates.

Division Algorithm

ALGORITHM D (division of nonnegative integers). Given nonnegative integers (u1, u2, . . . , ul) radix b, l=m+n, and (v1, v2, . . . , vn) radix b, where v1 != 0 and n > 1, we form the quotient floor (U/V) = (q1, q2, . . . , qm) radix b and the remainder U mod V = (r1, r2, . . . , rn) radix b.

D1. [Normalize] Set d = floor(b/(v1+1)). Then set (u0, u1, . . . , ul) radix b, l=m+n, equal to (v1, v2, . . . , vn) radix b times d. (Note the introduction of a new digit position u0 at the left of u1; if d=1, all we need to do in this step is to set u0=0. On a binary computer it may be preferable to chose d to be a power of two instead of using the value suggested here; any value of d that results in v1 >= floor(b/2) will suffice.)

D2. [Initialize j] Set j=0. (The loop on j, Steps D2 through D7, will be essentially a division of (uj, . . . , ur) radix b, r=j+n, by (v1, . . . , vn) radix b to get a single quotient digit qj.)

*D3. [Calculate q'] If uj=v1, set q'=b-1; otherwise set q' = floor((uj*b+ur)/ v1), r=j+1. Now test if v2*q' > (uj*b+ur-q'*v1)*b+us, r=j+1 and s=j+2; if so, decrease q' by 1 and repeat this test. (The latter test determines at high speed most of the cases in which the trial value q' is one too large, and it eliminates all cases where q' is two too large.)*

*D4. [Multiply and subtract] Replace (uj, . . . , ur) radix b, r=j+n, by (uj, . . . , ur) radix b minus q' times (v1, . . . , vn) radix b. This step (analogous to Steps M3, M4, and M5 above) consists of a simple multiplication by a one-place number, combined with a subtraction. The digits (uj, . . . , ur) radix b should be kept positive; if the result of this step is actually negative, (uj, . . . , ur) radix b should be left as the true value plus b ** (n+1) (i.e., as the b's complement of the true value) and a borrow to the left should be remembered.*

D5. [Test remainder] Set qj =q'. If the result of Step D4 was negative, go to Step D6; otherwise go to Step D7.

D6. [Add back] (The probability that this step will be necessary is very small, on the order of only 2/b; test data that activate this step should therefore be specifically contrived when debugging.) qj=qj-1, and add (0, v1, . . . , vn) radix b to (uj, . . . , ur) radix b, r=j+n. (A carry will occur to the left of uj, and it should be ignored since it cancels with the borrow that occurred in D4.)

D7. [Loop on j] j=j+1. Now if j <= m, go back to D3.

D8. [Un-normalize] Now (q0, q1, . . . , qm) radix b is the desired quotient, and the desired remainder may be obtained by dividing (ur, . . . , us) radix b, r=m+1 and s=m+n, by d.

As you can see, the implementation of these algorithms in Listing Four follows very closely, with the exception of some differences or additions that were required to make things work properly. A few areas are also identified in the listing where additions to the algorithms can be implemented. Obviously, these algorithms are very computation-intensive, and they form the root of the rest of the software. If we were to expend effort to optimize these routines, a real savings in execution time could undoubtedly be realized. A possibility would be to change the number arrays from byte-types to integer-types and to increase the arithmetic modulus MODULO. However, this change would create additional complications in handling text files because character packing would now have to be done. Another method to speed up the operations would be to use an arithmetic coprocessor, e.g., an AMD 9511 or an Intel 8087. Speed increases of up to an order of magnitude could potentially be realized.

We are now ready to discuss the final mathematical algorithm in this part of the article, the "Russian Peasant" algorithm. It will use the multiple-precision arithmetic algorithms just presented.

Russian Peasant Exponentiation

This algorithm was given its name by nineteenth-century visitors to Russia who found the technique in wide use there. However, the algorithm appeared in the fifteenth century and is based on a multiplication technique developed by the Egyptians at least 3300 years earlier. In certain aspects, the algorithm is similar to a binary search. The algorithm, as stated by Knuth, follows.

Russian Peasant Algorithm

*ALGORITHM A (right-to-left binary method for exponentiation). This algorithm evaluates X**N, where N is a positive integer.*

A1. [Initialize] Set n=N, y=1, z=X.

*A2. [Halve n] (At this point X**N = y * (z**n).) Determine whether n is odd or even. Set n=floor(n/2). If n is even, go to Step A5.*

*A3. [Multiply y by z] Set y = y * z.*

A4. [Check n for 0] If n = 0, the algorithm terminates with y as the answer.

*A5. [Square z] Set z = z * z, and go to Step A2.*

It should be pointed out that a dual algorithm for multiplication (y = X * N) can be achieved by replacing y=1 with y=0 in Step A1; by replacing Step A3 with an addition, i.e., y = y + z; and by replacing Step A5 with a doubling instead of a squaring, i.e., z = z + z. J. Nyberg has presented a BASIC program to perform this algorithm.[11] This method of exponentiation is usually suitable only for large exponents, N, since multiple-precision multiplication is more efficient for small and moderate values of N. Some faster methods are available. One involves the factoring of N into its prime factors. Another method uses a technique called "power trees." These methods are explained in more detail in Knuth's book, but these techniques are more complex than the Russian Peasant method.

The Russian Peasant algorithm had to be broadened to perform Y = (X ** N) mod M. The modulo operation limits the resultant answer to 0 <= Y < M. We shall see in the next part of the article that this form of the algorithm is required. The modification to the algorithm is as follows:

A5. [Square z] Set z = z * z.

A6. [Limit z] Set z = (z mod M) and go to A2.

The implementation of the expanded algorithm (RPEASANT.RAT) is found in Listing Five (page 38). The listing contains three subroutines:

RPEXP – The modified Russian Peasant algorithm.

PRDMOD – The routine that performs (Z * Z) mod M. It uses the multiple-

precision multiplication and division routines given in Listing Four.

MAKBIN – This routine tries to make the halving of the exponent, N, somewhat more efficient. The byte modulus, MODULO, does not have to be 2 ** K, 0 < K <= 7; thus, N cannot be strictly scanned on a bit-by-bit basis to perform the even/odd check and halving. So we must find a base modulus so that BASMOD = 2 ** ceiling (log2(MODULO)), and use it to strip off bytes that can be scanned on a bit-by-bit basis in the even/odd check. The operation becomes:

1. Initially, set Nx = N (first pass through)
2. TEMP = remainder(Nx/BASMOD)
3. Nx = floor(Nx / BASMOD) = quotient(Nx / BASMOD)

where TEMP is used by RPEXP to do Step A2 and the new Nx is retained for the next pass until Nx = 0. If MODULO = 2 ** K, TEMP just returns the current least-significant byte of Nx without doing the division. Therefore, it is most efficient to make MODULO = member{2,4,8,16,32,64, 128}.

As you can see in the multiple-precision routines and in the Russian Peasant routine, a considerable amount of pointer manipulation occurs. These routines probably could be implemented more efficiently in C or in Pascal, where pointer manipulation is more easily attained. I do not claim that these implementations are optimal; however, they do work. If you feel the need to "optimize" the code, dig in!

Summary

We are now at the end of the first part of the article. We have covered several areas: RATFOR, modulo arithmetic, multiple-precision arithmetic, and Russian Peasant exponentiation. If you implement these algorithms on your system, you should be able to check the addition and subtraction algorithms fairly easily. To check out the multiplication, division, and exponentiation algorithms, you will need a calculator with a large digit accuracy (>= 10 is desirable) or a copy of MUMATH or TK!SOLVER. Try this problem:

D = (A ** B) mod C

using

A = 9182736450
B = 1928374605
C = 12345678907
Byte Modulo = 100

to obtain

D = 10447988731

In the second and final part, we will look at the RSA-PKS encryption/decryption implementation. This will cover three additional areas: the generation and testing of large prime numbers, the generation of the public/private keys, and the encryption/decryption of text files with a brief look at digital signatures. See you on the second pass.

References

1 W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans Info Theory,* Vol. IT-22, No. 6, pp. 644-654.

2 M.E. Hellman, "The Mathematics of Public Key Cryptography," *Sci Amer,* February 1979, Vol. 241, No. 2, pp. 146-157.

3 R.L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Comm ACM,* February 1978, Vol. 21, No. 2, pp. 120-126.

4 S.G. Akl, "Digital Signatures: A Tutorial Survey," *Computer,* February 1983, Vol. 16, No. 2, pp. 15-24.

5 D.W. Davies, "Applying the RSA Digital Signature to Electronic Mail," *Computer,* February 1983, Vol. 16, No. 2, pp. 55-62.

6 R. DeMillo and M. Merritt, "Protocols for Data Security," *Computer,* February 1983, Vol. 16, No. 2, pp. 39-50.

7 D.E. Denning, "Protecting Public Keys & Signature Keys," *Computer,* February 1983, Vol. 16, No. 2, pp. 27-35.

8 B.W. Kernighan and P.J. Plauger, *Software Tools,* Reading, MA: Addison-Wesley, 338 pp.

9 D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* 2nd ed., Reading, MA: Addison-Wesley, pp. 250-268, 374-380, 386-389, 442-443.

10 M. Zimmermann, "Big Numbers & Small Computers," *Creative Computing,* January 1982, pp. 126-138.

11 J. Nyberg, "A Fast, Ancient Method of Multiplication," *Byte,* October 1981, pp. 376-377.

Additional References

1. B. Schanning, "Securing Data Inexpensively Via Public Keys," *Computer Design,* April 5, 1983, pp. 105-108.
2. J. Smith, "Public Key Cryptography," *Byte,* January 1983, pp. 198-218.
3. H. T. Gordon, "Fast Divisibility Algorithms," *DDJ,* June 1983, No. 80, pp. 14-16. ■■J

RSA Cryptography System (Text begins on page 16)

Listing One

```
# *** Listing 1 -- an example of the RATFOR precompiler output
# ***

#              Note that the resulting FORTRAN has the spaces and new lines
#              deleted and that the result is much less readable than the
#              RATFOR source !!!

define(NEWVARIABLE,variable30) # look at Label 8 for the replacement

program example

      integer variable1,variable2,variable3,variable4,variable5,variable6,
             variable7,variable8,variable9,variable10

1     continue # separator for use in finding where statements begin in
               #  generated FORTRAN code (standard FORTRAN statement)
      repeat # a RATFOR Construct
           { # "{" and "}" are not needed for a single statement (see WHILE) !!!
           statement a
           }
      until (variable1 == variable2)

2     continue
      while (variable3 != variable4)
           statement b # in-line comment

3     continue
      for (variable5=0.0; variable5 < variable6; variable5=variable5+1.0)
           {
           statement c
           if (variable7 >= variable8)
                {
                break
                }
           }

4     continue
      do n=j,k  # Note the lack of a terminating label number.  RATFOR
                #  will fill it in for you !!!
           {
           if (variable9 >= variable10)
                {
                next
                }
           statement d
           }
5     continue
      if (variable7 > variable8)
          { # "{" and "}" must be used here since ";" makes multiple lines !!!
          statement e; statement f; statement g
          }
     else if ((! logical1) & (variable9 <= variable10) |
               (logical2 & logical3))
          {
```

```
        statement h
        }
    else
        {
        statement i
        }

include LISTING2 # insert Listing 2 here
```

End Listing One

Listing Two

```
# *** Listing 2 -- this file will be included into Listing 1 at the occurrance
# ***               of the "include LISTING2" statement !!!


6     call subroutine(variable11,variable12,variable13,variable14,variable15,
                      variabl116,variable17,variable18,variable19,variable20)


7     variable21=variable22*variable23/variable24+variable25-    _
                (variable26/variable27)+variable28


8     variable31=NEWVARIABLE
```

End Listing Two

Listing Three

```
C *** Listing 3 -- resultant FORTRAN source generated by passing
C ***               Listing1 & Listing2 through RATFOR
C ***
      programexample
      integervariable1,variable2,variable3,variable4,variable5,variable6
     *,variable7,variable8,variable9,variable10
1     continue
      continue
23000 continue
      statementa
23001 if(.not.(variable1.eq.variable2))goto 23000
23002 continue
2     continue
      continue
23003 if(.not.(variable3.ne.variable4))goto 23004
      statementb
      goto 23003
23004 continue
3     continue
      continue
      variable5=0.0
```

(Continued on page 26)

RSA Cryptography System (Listing continued, text begins on page 16)

Listing Three

```
23005 if(.not.(variable5.lt.variable6))goto 23007
      statementc
      if(.not.(variable7.ge.variable8))goto 23008
      goto 23007
23008 continue
23006 variable5=variable5+1.0
      goto 23005
23007 continue
4     continue
      do 23010n=j,k
      if(.not.(variable9.ge.variable10))goto 23012
      goto 23010
23012 continue
      statementd
23010 continue
23011 continue
5     continue
      if(.not.(variable7.gt.variable8))goto 23014
      statemente
      statementf
      statementg
      goto 23015
23014 continue
      if(.not.((.not.logical1).and.(variable9.le.variable10).or.(logical
     *2.and.logical3)))goto 23016
      statementh
      goto 23017
23016 continue
      statementi
23017 continue
23015 continue
6     callsubroutine(variable11,variable12,variable13,variable14,variabl
     *e15,variabl16,variable17,variable18,variable19,variable20)
7     variable21=variable22*variable23/variable24+variable25-(variable26
     */variable27)+variable28
8     variable31=variable30
      end
```

End Listing Three

Listing Four

```
###############################################################################
###                                                                         ###
###        Copyright 1983, Charles E. Burton, Denver, Colorado              ###
###                                                                         ###
### All rights reserved.  Permission granted to use this software for       ###
###   personal, non-commercial purposes only.                               ###
###                                                                         ###
###############################################################################

# PROGRAM NAME:  MPARITH.RAT
# PURPOSE:  Unsigned Multiple Precision Arithmetic Routines (re: D.E. Knuth,
#           The Art of Computer Programming, V. 2 (Semi-Numerical Algorithms),
#           2nd Ed., (Addison-Wesley, Reading, MA), pp. 250-268.)
#                   MPADD  -- Multiple Precision Addition
#                   MPSUBT -- Multiple Precision Subtraction
#                   MPMULT -- Multiple Precision Multiplication
#                   MPDIV  -- Multiple Precision Division
```

```
#
# LANGUAGE:  RATFOR
# AUTHOR:  CEB
# USAGE:  CALL MPADD(NUM1,LEN1,NUM2,LEN2,SUM,LENS,MODULO)
#         CALL MPSUBT(NUM1,LEN1,NUM2,LEN2,DIFF,LEND,MODULO)
#         CALL MPMULT(NUM1,LEN1,NUM2,LEN2,PROD,LENP,MODULO)
#         CALL MPDIV(NUM1,LEN1,NUM2,LEN2,QUOT,LENQ,REMN,LENR,MODULO)
#              <>NUM* -- Byte array, contains the number (byte modulus MODULO)
#                        to be operated on. The MSDigit(s) are in
#                        NUM*(1) and the LSDigit(s) are in NUM*(LEN*).  NUM2 is
#                        never modified and NUM1 is only modified by MPDIV.
#                        CAUTIONS:  for MPSUBT, NUM1 must be the larger number;
#                        for MPDIV, NUM1 must be the Numerator such that
#                        NUM1(1) == 0 (required for normalization) and NUM2
#                        must be the Demominator such that NUM2(1) != 0.
#              >LEN1/2 -- Integer variable, defines the length of the NUM*
#                         array.
#              <SUM -- Byte array, contains the sum (byte modulus MODULO)
#                      of NUM1 + NUM2.  The MSDigit(s) are in
#                      SUM(1) and the LSDigit(s) are in SUM(LENS).
#              >LENS -- Integer variable, defines length of SUM array.
#                       CAUTION:  LENS = MAX(LEN1,LEN2) + 1
#              <DIFF -- Byte array, contains the difference (byte modulus
#                       MODULO) of NUM1 - NUM2.  The MSDigit(s) are in
#                       DIFF(1) and the LSDigit(s) are in DIFF(LEND).
#              >LEND -- Integer variable, defines length of DIFF array.
#                       CAUTION:  LEND = LEN1
#              <PROD -- Byte array, contains the product (byte modulus MODULO)
#                       of NUM1 * NUM2.  The MSDigit(s) are in
#                       PROD(1) and the LSDigit(s) are in PROD(LENP).
#              >LENP -- Integer variable, defines length of PROD array.
#                       CAUTION:  LENP = LEN1 + LEN2
#              <QUOT -- Byte array, contains the quotient (byte modulus MODULO)
#                       of NUM1 / NUM2.  The MSDigit(s) are in
#                       QUOT(1) and the LSDigit(s) are in QUOT(LENQ).
#              >LENQ -- Integer variable, defines length of QUOT array.
#                       CAUTION:  LENQ = MAX(LEN1 - LEN2, 1)
#              <REMN -- Byte array, contains the remainder (byte modulus
#                       MODULO) of NUM1 / NUM2.  The MSDigit(s) are in
#                       REMN(1) and the LSDigit(s) are in REMN(LENR).
#              >LENR -- Integer variable, defines length of REMN array.
#                       CAUTION:  LENR = LEN2
#              >MODULO -- Integer variable, defines the arithmetic modulus that
#                         is to be used.  MODULO has a byte-wide effect and
#                         should be between 2 and 128 (e.g. 100 for Decimal
#                         Numbers and 128 for ASCII Characters).
#                         CAUTION:  The Arrays MUST have the same modulus and
#                         MUST be positive (unsigned) for proper operation !!!
#
#
# ARRAYS USED: NUM1(*),NUM2(*),SUM(*),DIFF(*),PROD(*),QUOT(*),REMN(*)
# EXTERNALS:
# UPDATE HISTORY:  INITIAL RELEASE -- 01/18/83 CEB
#

subroutine mpadd(num1,len1,num2,len2,sum,lens,modulo)

     byte num1(1),num2(1),sum(1)
     integer len1,len2,lens,modulo,add,carry
```

(Continued on next page)

RSA Cryptography System (Listing continued, text begins on page 16)

Listing Four

```
    if (lens >= max0(len1,len2)+1) # SUM array have sufficient length?
          {
          idxsum=lens # get the index to the LSDigit of SUM
          idxn1=len1 # get index to LSDigit of NUM1
          idxn2=len2 # get index to LSDigit of NUM2
          carry=0 # initialize CARRY
          while (min0(idxn1,idxn2) > 0) # look at each packed character
                {
                itemp1=num1(idxn1); itemp2=num2(idxn2) # byte to integer
                add=carry+itemp1+itemp2 # get sum
                sum(idxsum)=mod(add,modulo) # generate SUM value
                carry=add/modulo # generate CARRY for next pass
                idxn1=idxn1-1 # move indices to next position
                idxn2=idxn2-1
                idxsum=idxsum-1
                }
          if (idxn2 == 0) # 1st number still has data?
                {
                while (idxn1 > 0) # look at rest of 1st number
                      {
                      itemp1=num1(idxn1) # byte to integer
                      add=carry+itemp1 # get sum
                      sum(idxsum)=mod(add,modulo) # generate SUM value
                      carry=add/modulo # generate CARRY for next pass
                      idxn1=idxn1-1 # move indices to next position
                      idxsum=idxsum-1
                      }
                }
           else # 2nd number still has data?
                {
                while (idxn2 > 0) # look at rest of 2nd number
                      {
                      itemp1=num2(idxn2) # byte to integer
                      add=carry+itemp1 # get sum
                      sum(idxsum)=mod(add,modulo) # generate SUM value
                      carry=add/modulo # generate CARRY for next pass
                      idxn2=idxn2-1 # move indices to next position
                      idxsum=idxsum-1
                      }
                }
          sum(idxsum)=carry # put final CARRY in SUM
          idxsum=idxsum-1 # move index next position
          while (idxsum > 0) # finish filling in SUM
                {
                sum(idxsum)=0 # zero out remainder of SUM
                idxsum=idxsum-1 # move index to next position
                }
          }
      else # cannot generate SUM
          {
          call rmrkln('.')
          call error('Length of [SUM] too small !!!.') # print message & exit
          }
      return
end

##############################################################################
```

```
subroutine mpsubt(num1,len1,num2,len2,diff,lend,modulo)

      byte num1(1),num2(1),diff(1)
      integer len1,len2,lend,modulo,subt,borrow

      if ((lend >= len1) & (len1 >= len2)) # DIFF array have sufficient length
                                           #  and NUM1 at least as big as NUM2?
            {
            idxdif=lend # get the index to the LSDigit of DIFF
            idxn1=len1 # get index to LSDigit of NUM1
            idxn2=len2 # get index to LSDigit of NUM2
            borrow=0 # initialize BORROW
            while (idxn2 > 0) # look at each packed character
                  {
                  itemp1=num1(idxn1); itemp2=num2(idxn2) # byte to integer
                  subt=borrow+itemp1-itemp2 # get difference
                  if (subt < 0) # need to do MODULO's complement?
                        {
                        subt=modulo+subt # do MODULO's complement
                        borrow=-1 # indicate borrow
                        }
                  else # everything is okay
                  borrow=0
            diff(idxdif)=subt # generate DIFF value
            idxn1=idxn1-1 # move indices to next position
            idxn2=idxn2-1
            idxdif=idxdif-1
            }
      while (idxn1 > 0) # look at rest of 1st number
            {
            itemp1=num1(idxn1) # byte to integer
            subt=borrow+itemp1 # get difference
            if (subt < 0) # need to do MODULO's complement?
                  {
                  subt=modulo+subt # do MODULO's complement
                  borrow=-1 # indicate borrow
                  }
            else # everything is okay
                  borrow=0
            diff(idxdif)=subt # generate DIFF value
            idxn1=idxn1-1 # move indices to next position
            idxdif=idxdif-1
            }
      if (borrow == 0) # NUM1 >= NUM2?
            {
            while (idxdif > 0) # finish filling in DIFF
                  {
                  diff(idxdif)=0 # zero out remainder of DIFF
                        idxdif=idxdif-1 # move index to next position
                        }
                  }
            else # NUM1 < NUM2 (error)
                  {
                  call rmrkln('.')
```

(Continued on next page)

Listing Four

```
                call error('[NUM1] < [NUM2] in subtraction.') # print message
                                                              #  & exit
                }
          }
     else # cannot generate DIFF
          {
          call rmrkln('.')
          call error('Length of [DIFF] too small or [NUM1] < [NUM2] !!!.')
                                                    # print message & exit
          }
     return

end
################################################################################

subroutine mpmult(num1,len1,num2,len2,prod,lenp,modulo)

     byte num1(1),num2(1),prod(1)
     integer len1,len2,lenp,modulo,mult,carry

     if (lenp >= len1+len2) # PROD array have sufficient length?
          {
          idxn2=len2 # get index to LSDigit of NUM2
          do idxprd=1,lenp # clean out PROD
               prod(idxprd)=0
          while (idxn2 > 0) # multiply numbers
               {
               idxprd=lenp+(idxn2-len2) # get index to PROD
               if (num2(idxn2) == 0) # multiplier zero?
                    {
                    carry=0 # set CARRY (product) to zero
                    idxprd=idxprd-len1 # get index to PROD
                    }
               else # finite multiplier value
                    {
                    idxn1=len1 # get index to LSDigit of NUM1
                    carry=0 # initialize CARRY
                    while (idxn1 > 0) # multiply a "digit" at a time
                         {
                         itemp1=num1(idxn1); itemp2=num2(idxn2) # byte to
                         itemp3=prod(idxprd)                    #  integer
                         mult=carry+itemp1*itemp2+itemp3 # get product
                         prod(idxprd)=mod(mult,modulo) # generate new product
                                                      #  value
                         carry=mult/modulo # generate CARRY for next pass
                         idxn1=idxn1-1 # move indices to next position
                         idxprd=idxprd-1
                         }
                    }
               prod(idxprd)=carry # save CARRY for next pass
                    idxn2=idxn2-1 # move index to NUM2 to next position
                    }
               }
```

```
      else # cannot generate PROD
           {
           call rmrkln('.')
           call error('Length of [PROD] too small !!!.') # print message & exit
           }
      return

end
################################################################################

subroutine mpdiv(num1,len1,num2,len2,quot,lenq,remn,lenr,modulo)

      byte num1(1),num2(1),quot(1),remn(1)
      integer len1,len2,lenq,lenr,modulo,div,mult,add,borrow,carry,flag,scale,
              qtest

      equivalence (borrow,carry,flag),(div,mult,add)

      if ((lenq >= len1-len2) & (lenr >= len2) & (num1(1) == 0) &
          (num2(1) != 0)) # QUOT & REMN arrays have sufficient length and
                          #  valid numerator & denominator?
           {
           do idxquo=1,lenq # clean out QUOT
                quot(idxquo)=0
           do idxrem=1,lenr # clean out REMN
                remn(idxrem)=0
           flag=0 # initialize numerator zero flag
           do idxn1=2,len1 # check numerator for Zero
                {
                if (num1(idxn1) != 0) # non-zero "digit" found?
                     {
                     flag=1 # indicate non-zero numerator
                     break # exit DO loop
                     }
                }
           #
           # could check for Numerator <= Denominator here !!!
           #    if (Numerator == Denominator)
           #         QUOT = 1
           #         REMN = 0
           #    else if (Numerator < Denominator)
           #         QUOT = 0
           #         REMN = Numerator
           #    else
           #         continue
           #
           if (flag != 0) # non-zero numerator?
                {
                itemp1=num2(1) # byte to integer
                scale=modulo/(itemp1+1) # get normalizing scale factor
                if (scale > 1) # normalization required?
                     {
                     carry=0 # initialize CARRY
                     idxn1=len1 # initialize numerator pointer
                     while (idxn1 > 0) # normalize numerator
```

(Continued on next page)

RSA Cryptography System (Listing continued, text begins on page 16)

Listing Four

```
        {
        itemp1=num1(idxn1) # byte to integer
        mult=carry+scale*itemp1 # get product
        num1(idxn1)=mod(mult,modulo) # generate new product
                                     #  value
        carry=mult/modulo # generate carry for next pass
        idxn1=idxn1-1 # move index to next position
        }
carry=0 # initialize CARRY

        idxn2=len2 # initialize denominator pointer
        while (idxn2 > 0) # normalize denominator
                {
                itemp1=num2(idxn2) # byte to integer
                mult=carry+scale*itemp1 # get product
                num2(idxn2)=mod(mult,modulo) # generate new product
                                             #  value
                carry=mult/modulo # generate carry for next pass
                idxn2=idxn2-1 # move index to next position
                }
        }
# else NUM1(1) == 0, so scaling is already done
idxn1=1 # initialize index of NUM1
while (idxn1 <= (len1-len2)) # calculate quotients
        {
        if (num2(1) == num1(idxn1)) # get initial test quotient
                qtest=modulo-1
        else
        {
        itemp1=num1(idxn1); itemp3=num2(1) # byte to integer
        if ((idxn1+1) <= len1) # valid index?
                itemp2=num1(idxn1+1)
        else # no
                itemp2=0
        qtest=(modulo*itemp1+itemp2)/itemp3
        }
itemp2=num1(idxn1); itemp4=num2(1) # byte to integer
if (len2 >= 2) # valid index?
        itemp1=num2(2)
else # no
        itemp1=0
if ((idxn1+1) <= len1) # valid index?
        {
        itemp3=num1(idxn1+1)
        if ((idxn1+2) <= len1) # valid index?
                itemp5=num1(idxn1+2)
        else # no
                itemp5=0
        }
else # no
        {
        itemp3=0
        itemp5=0
        }
while (qtest*itemp1 > (modulo*(modulo*itemp2+itemp3-
            qtest*itemp4)+itemp5))
                # check if test quotient is too large
```

```
      {
      qtest=qtest-1
      }
idxn2=len2 # initialize index to NUM2
idx=idxn1+len2 # initialize index to NUM1
borrow=0 # initialize BORROW



while (idx >= idxn1) # perform multiply & subtract
      {
      if (idxn2 > 0) # another "digit" of NUM2?
            {
            itemp1=num2(idxn2) # byte to integer
            borrow=borrow-qtest*itemp1
            }
      itemp1=num1(idx) # byte to integer
      div=borrow+itemp1 # generate mult. & subt.
      if (div < 0) # need to do MODULO's complement?
            {
            borrow=div/modulo # get BORROW for next pass
            div=mod(div,modulo) # MODULO's complement
            if (div < 0) # still need to make DIV positive?
                  {
                  div=modulo+div # make it positive
                  borrow=borrow-1 # adjust borrow
                  }
            }
      else # everything is okay
            borrow=0
      num1(idx)=div # update numerator
      idxn2=idxn2-1 # move indices to next position


      idx=idx-1
      }
idxquo=lenq-(len1-len2)+idxn1 # get index to QUOT
if (borrow != 0) # need to add back divisor?
      {
      quot(idxquo)=qtest-1 # adjust quotient
      idxn2=len2 # get index to NUM2
      idx=idxn1+len2 # get index to NUM1
      carry=0 # initialize CARRY
      while (idx >= idxn1) # add back
            {
            if (idxn2 > 0) # another "digit" of NUM2?
                  {
                  itemp1=num2(idxn2) # byte to integer
                  carry=carry+itemp1 # add it back
                  }
            itemp1=num1(idx) # byte to integer
            add=carry+itemp1 # add it back in
            if (add >= modulo) # need to adjust ADD?
                  {
                  add=add-modulo # pull ADD back into range
                  carry=1 # set CARRY for next pass
                  }
            else # no adjustment
                  carry=0 # no carry
            num1(idx)=add # update numerator
```

(Continued on next page)

RSA Cryptography System (Listing continued, text begins on page 16)

Listing Four

```
                    idxn2=idxn2-1 # move indices to next position
                    idx=idx-1
                    }
               }
                    else # no add back needed
                         quot(idxquo)=qtest # generate quotient
                    idxn1=idxn1+1 # move index to next position
                    }
               idxn1=len1-len2+1 # get index to NUM1
               idxrem=lenr-len2+1 # get index to REMN
               borrow=0 # initialize BORROW
               while (idxn1 <= len1) # unnormalize remainder
                    {
                    itemp1=num1(idxn1) # byte to integer
                    div=itemp1+modulo*borrow # get numerator of REMN
                    remn(idxrem)=div/scale # remove normalization
                    borrow=mod(div,scale) # get borrow for next pass
                    idxn1=idxn1+1 # move indices to next position
                    idxrem=idxrem+1
                    }
               idxn2=1 # get index to NUM2
               borrow=0 # initialize BORROW
               while (idxn2 <= len2) # unnormalize NUM2
                    {
                    itemp1=num2(idxn2) # byte to integer
                    div=itemp1+modulo*borrow # get numerator of NUM2
                    num2(idxn2)=div/scale # remove normalization
                    borrow=mod(div,scale) # get borrow for next pass
                    idxn2=idxn2+1 # move index to next position
                    }
               }
          }
     else # cannot generate QUOT & REMN
          {
          call rmrkln('.')
          call remark('Length of [QUOT] or [REMN] too small or .')
          call error('invalid numerator or denominator !!!.')
                    # print message and exit
          }
     return

end
```

End Listing Four

Listing Five

```
###############################################################################
###                                                                         ###
###        Copyright 1983, Charles E. Burton, Denver, Colorado              ###
###                                                                         ###
### All rights reserved.  Permission granted to use this software for       ###
###   personal, non-commercial purposes only.                               ###
###                                                                         ###
###############################################################################

# PROGRAM NAME:  RPEASANT.RAT
# PURPOSE:  Russian Peasant exponentiation (re. D.E. Knuth,
```

```
#           The Art of Computer Programming, V. 2 (Semi-Numerical Algorithms),
#           2nd Ed., (Addison-Wesley, Reading, MA), pp. 442-443.)
#
#
# LANGUAGE:  RATFOR
# AUTHOR:  CEB
# USAGE:  CALL RPEXP(NUM1,LEN1,NUM2,LEN2,EXPN,LENE,MODL,LENM,WORK,LENW,MODULO)
#               <>NUM* -- Byte array, contains the number [NUM* mod MODL] (byte
#                         modulus MODULO) to be operated on. The MSDigit(s) are
#                         in NUM*(1) and the LSDigit(s) are in NUM*(LEN*).
#                         NUM2 is never modified, but NUM1 is always modified !
#               >LEN1/2 -- Integer variable, defines the length of the NUM*
#                          array.
#                          CAUTION:  LEN1 = LENM
#               <EXPN -- Byte array, contains the exponentiation (byte modulus
#                        MODULO) of [NUM1 ** NUM2 mod MODL].  The MSDigit(s) are
#                        in EXPN(1) and the LSDigit(s) are in EXPN(LENE).
#               >LENE -- Integer variable, defines length of EXPN array.
#                        CAUTION:  LENE = MAX(LEN1, LENM)
#               >MODL -- Byte array, contains the modulus of the arithmetic to
#                        be used in the calculations (byte modulus MODULO).
#                        The MSDigit(s) are in MODL(1) and the LSDigit(s) are
#                        in MODL(LENM).
#                        CAUTION:  MODL(1) != 0
#               >LENM -- Integer variable, defines length of MODL array.
#               <>WORK -- Byte array, a working array needed to do modulus
#                         arithmetic using MODL, i.e. [ ? mod MODL].
#               >LENW -- Integer variable, defines length of WORK array.
#                        CAUTION:  LENW = (LEN2 + 1)                <NUM2 temp.>
#                                       + (LEN1 + LENE) + 1         <Numerator>
#                                       + MAX(LENE - LENM, 1)       <Quotient>
#                                       + LENM                      <Remainder>
#               >MODULO -- Integer variable, defines the arithmetic byte modulus
#                          that is to be used.  MODULO has a byte-wide effect
#                          and should be between 2 and 128 (e.g. 100 for Decimal
#                          Numbers and 128 for ASCII Characters).
#                          CAUTION:  The Arrays MUST have the same modulus and
#                          MUST be positive (unsigned) for proper operation !!!
#
#
# ARRAYS USED: NUM1(*),NUM2(*),EXPN(*),MODL(*),WORK(*)
# EXTERNALS:  MPMULT,MPDIV
# UPDATE HISTORY:  INITIAL RELEASE -- 01/20/83 CEB
#
subroutine rpexp(num1,len1,num2,len2,expn,lene,modl,lenm,work,lenw,modulo)

      byte num1(1),num2(1),expn(1),modl(1),work(1)
      logical maskit,bit,temp,nextf
      integer len1,len2,lene,lenm,lenw,modulo,mask,basmod

      if ((len1 >= lenm) & (lene >= max0(len1,lenm)) &
          (lenw >= (len2+1)+(len1+lene+1)+max0(lene-lenm,1)+lenm))
                            # NUM1, EXPN & WORK arrays have sufficient lengths?
           {
           basmod=1 # initialize basic modulus
           repeat # find basic modulus
                basmod=2*basmod # advance basic modulus
           until ((basmod == 128) | (modulo/basmod == 1)) # basic modulus found
           do idxexp=1,lene # initialize EXPN array
                expn(idxexp)=0
           expn(lene)=1 # set EXPN=1
```

(Continued on next page)

RSA Cryptography System (Listing Continued, text begins on page 16)

Listing Five

```
      do idxn2=1,len2 # copy NUM2 to temporary NUM2
           work(idxn2+1)=num2(idxn2)
      work(1)=0 # make sure it starts with zero (for MAKBIN routine)
      idxn2=len2 # get index to NUM2
      nextf=.true. # initialize next pass flag
      while (nextf) # generate Russian Peasant exponential
           {
           call makbin(work,idxwrk,temp,basmod,modulo,len2,idxn2,nextf)
                                                # make temporary NUM2 binary
           mask=1 # initialize bit mask
           while (mask < basmod) # run through bits within each byte
                                 #  of NUM2
                {
                maskit=mod(mask,256) # get LSByte of MASK
                bit=temp & maskit # get appropiate bit of NUM2
                if (bit != 0) # need to multiply NUM1 by EXPN?
                     call prdmod(expn,lene,num1,len1,modl,lenm,
                                 work(idxwrk),modulo)
                               # generate EXPN = (EXPN*NUM1) mod MODL
                call prdmod(num1,len1,num1,len1,modl,lenm,work(idxwrk),
                            modulo) # generate NUM1 = (NUM1*NUM1) mod MODL
                mask=2*mask # move to next mask bit
                }
                idxn2=idxn2-1 # move to next NUM2 Digit
                }
           }
      else # cannot generate Russian Peasant Product
           {
           call rmrkln('.')
           call error('Length of [NUM1], [EXPN] or [WORK] too small !!!.')
                # print message & exit
           }
      return

end
################################################################################

# PRDMOD -- Product Modulo Routine:  NUM1 = (NUM1*NUM2) mod MODL

subroutine prdmod(num1,len1,num2,len2,modl,lenm,work,modulo)

      byte num1(1),num2(1),modl(1),work(1)
      integer len1,len2,lenm,modulo

      lenp=len1+len2 # initialize length of product of NUM1 & NUM2
      call mpmult(num1,len1,num2,len2,work,lenp,modulo) # multiply NUM1 & NUM2
      if (work(1) != 0) # need to shift result for normalization in Division?
           {
           idxsum=lenp # initialize index to product
           while (idxsum > 0) # shift result down 1 byte
                {
                work(idxsum+1)=work(idxsum)
                idxsum=idxsum-1 # move index to next position
                }
           work(1)=0 # set MSDigit to zero
           lenp=lenp+1 # advance sum length
           }
      idxquo=lenp+1 # initialize pointer to start of Quotient array
```

```
      lenq=max0(lenp-lenm,1) # get Quotient length
      idxrem=idxquo+lenq # initialize pointer to start of Remainder array
      call mpdiv(work,lenp,modl,lenm,work(idxquo),lenq,work(idxrem),lenm,modulo)
                  # generate (NUM1*NUM2) mod MODL
      idxend=idxrem+lenm-1 # initialize index to end of remainder
      idxn1=len1 # get index to NUM1
      while (idxend >= idxrem) # remainder digits are available
            {
            num1(idxn1)=work(idxend) # move remainder to NUM1
            idxn1=idxn1-1 # move indices to next position
            idxend=idxend-1
            }
      while (idxn1 > 0) # any Digits left in NUM1
            {
            num1(idxn1)=0 # zero it
            idnx1=idxn1-1 # move index to next position
            }
      return
end
################################################################################
# MAKBIN -- make sure Temporary NUM2 is binary
subroutine makbin(work,idxwrk,temp,basmod,modulo,len2,idxn2,nextf)
      byte work(1),bmod
      logical temp,nextf
      integer idxwrk,basmod,modulo,len2,idxn2

      lenw=len2+1 # get length of work
      nextf=.false. # initialize next pass flag
      if (basmod != modulo) # NUM2 modulus not binary?
            {
            idxquo=lenw+1 # get index to quotient
            lenq=len2 # get length of quotient
            idxrem=idxquo+lenq # get index to remainder
            bmod=basmod # integer to byte
            call mpdiv(work,lenw,bmod,1,work(idxquo),lenq,work(idxrem),1,modulo)
                                                # make temp. NUM2 binary
            temp=work(idxrem) # get binary temporary NUM2
            idxquo=idxquo+lenq-1 # get index to quotient
            idxwrk=lenw # get index to work
            while (idxquo > lenw) # move quotient to temporary NUM2 for next pass
                  {
                  if (work(idxquo) != 0) # still digits left?
                        nextf=.true. # flag a next pass
                  work(idxwrk)=work(idxquo)
                  idxquo=idxquo-1 # move indices to next position
                  idxwrk=idxwrk-1
                  }
            while (idxwrk > 0) # zero out remainder of temporary NUM2
                  {
                  work(idxwrk)=0
                  idxwrk=idxwrk-1 # move index to next position
                  }
            }
      else # NUM2 modulus is binary
            {
            if (idxn2 > 1) # still digits left?
                  nextf=.true. # flag a next pass
            temp=work(idxn2+1) # get binary temporary NUM2
            }
      idxwrk=lenw+1 # get index to start of unused work area
      return
end
```

End Listing

Introduction to PL/C:
Programming Language for Compilers

by Morris Dovey

Morris R. Dovey, MRD Systems Incorporated, P. O. Box 147, Spring Valley, MN 55975.

Copyright © 1983 Morris R. Dovey. All rights reserved.
License is granted for personal and non-commercial use of this material. All commercial rights are reserved.

As LSI densities and chip yields increased steadily, it became apparent that our ability to produce new and more powerful processors exceeded our ability to produce supporting software. An examination of technology advances indicates that hardware gains are due as much to the development of an appropriate tool technology as to the amount of direct resources expended. Efforts in the software domain, however, appear to have been directed much more toward producing immediately usable results than toward advancing the state of the art; the problem is made worse by the high cost of the talent required to implement system software.

One solution, seen by many quite early in the game, was to make software that could be transported from one processor to another in either of two ways: The machine code could be transported, or high-level language source programs could be transported and recompiled for the new machine.

The first approach works well as long as the machine codes remain constant from processor to processor and the destination processor operates in a manner functionally equivalent to the source processor. The primary drawback is a (perhaps unhappy) marriage to a fixed set of operations and instruction formats. The advantages and drawbacks are exemplified by IBM's 360 and 370 series mainframes. That architecture is now over fifteen years old. Its preservation has allowed the transporting of code from model to model, but it has also inhibited technological growth.

The second approach to portability works well provided that the high-level language facility exists for the destination processor and that the high-level language compilers produce functionally equivalent object modules. The primary disadvantage of this approach is that the required language facility often arrives well after the processor. Also, as a consequence of normal system software development (wherein we keep reinventing the same old wheel), the language processors frequently do not produce functionally equivalent object modules.

"It is helpful to think of PL/C as a strictly structured and totally procedural programming language. . . ."

The PL/C language has been developed to address the problems inherent in the second approach. PL/C makes possible the implementation of compiler core programs in a processor-independent form, enabling compilers to be developed with much greater speed than before. The PL/C compiler program has itself been implemented in a processor-independent form. Listing One (page 50) is the PL/C listing for the compiler itself. It requires approximately 24 subroutines (a total of about 2K bytes to complete its implementation for any given processor. Listing Two (page 54) is the compiler in macro-assembler code.

PL/C Structural Notation

A PL/C program consists of a sequence of labeled statements, followed by an unlabeled statement consisting of the word "end" (in either upper or lower case), followed by a semicolon. For convenience we will set the end statement aside as a special case and refer to labeled statements simply as "statements."

Statements begin with a label, which may be followed by any number of PL/C specification items, and end with a semicolon. A label consists of a PL/C statement name followed by a colon (or "::="). A statement name consists of a left angle bracket (<), followed by the name, followed by a right angle bracket (>). The name itself may contain imbedded blanks.
within a PL/C statement as a specification item, there must be a statement labeled with the same name. Obviously, syntax definition must continue until all syntactical elements have been resolved to literal values.

Ordering of statements is arbitrary, except that the first statement must be the highest level of definition used. PL/C syntax is fully defined in Table I (page 47).

PL/C Program Structure and Flow

It is helpful to think of PL/C as a strictly structured and totally procedural programming language in which the first statement of any program is the main procedure and all subsequent statements are subprocedures. That way, all syntax item specifications within the body of each statement may be regarded as subprocedure calls. The control flow map of any PL/C program, then, is a tree structure at every node of which is a PL/C specification item.

A node may or may not be classified as "output producing." Output may consist of data collected from the input string undergoing compilation, or it may be data generated explicitly by an output specification. The PL/C language provides for the ordered passing of output data up the tree structure from node to node. The output data passed all the way up to the initial node becomes "actual" output: the result of the compilation process. Operationally, then, a compiler programmed in PL/C is not much different from most other compilers. The real differences lie in the development process.

PL/C program flow control is entirely implicit: no language facility exists for explicit transfer of control. Since most programming errors are flow related, this strictly structured approach minimizes many common errors. The problem of attempting to enforce internal structured-programming standards is avoided: not only does the language itself make the use of explicit flow control statements seem unnatural, but it also de-emphasizes the entire concept of "flow" by dispensing with the usual notion that control passes from statement to statement in the order in which the statements appear. In addition, the structured approach makes it easy to partition the development effort and removes most of the difficulties involved in follow-on maintenance and enhancement.

The BNF (Bakus-Naur Format) description of the target language constitutes the bulk of the compiler code. Because published BNF descriptions are readily available for every major programming language in use, the entire scanning logic for a compiler in effect is available on an "off the shelf" basis. This allows the development effort to be concentrated where it belongs: in the generation of correct and efficient compiler output.

Syntactical Elements

The PL/C programming language is a natural extension of the BNF notation for language syntax definition and description. PL/C extends BNF in three essential ways:

1. An output facility has been provided.
2. Input collection and symbol manipulation facilities have been added.
3. A facility has been included to permit in-line inclusion of instructions and/or data for processing at assembly time.

Some changes of notation from BNF are necessary. Character literals in BNF are normally undelimited; in PL/C they are delimited by single quotes in order to deal with the space character.

The asterisk (*), used to denote the optional occurrence of a specification, is placed following the optional item specification. For example:

```
<procedure keyword>::=
    'PROC' 'EDURE'*;
```

indicates that the character string "PROC" is required at the current point in the input file being scanned. Once this requirement is satisfied, the string "EDURE" is permitted but not required. Thus the specification "procedure keyword" may be satisfied by either the string "PROC" or the string "PROCEDURE."

The dollar sign ($) is used to denote the occurrence of a specification one or more times to satisfy the containing specification. An example of its use might be:

```
<integer>::= <decimal digit> $;
```

where the specification "integer" will be satisfied if "decimal digit" is satisfied one or more times.

The dollar sign and asterisk may be combined to indicate that many optional occurrences of the specification are permitted. For example:

```
<skip>::= ' '$*;
```

will cause the compiler to scan to the next nonblank character in the input stream. If the character at the current point in the input stream is not a space character, the scan will not advance; however, the "skip" specification will not fail since the space(s) are only permitted, not required.

The vertical bar (|) is used to indicate that, should the preceding specification fail, the following specification is to be accepted by the compiler as a valid alternative. For example:

```
<arithmetic atom>::=
    <constant> | <variable>;
```

indicates that the specification "arithmetic atom" will be satisfied by the success of either the "constant" specification or the "variable" specification. If the "constant" specification is satisfied, the "variable" specification will not be checked.

All statements are terminated with a semicolon. PL/C programs use a free format, which means that a specification (statement) may extend over as many lines as desired and that several statements may be coded in a single line; in short, line (record) boundaries are ignored.

PL/C Output Facilities

Output specifications in PL/C may occur anywhere within the containing specification and are enclosed in brackets ([ ]). Three modes of output are available under the current implementation:

Character string output

Hexadecimal value byte output

Subordinate node output

Character string output is accomplished by enclosing the string to be generated in single quotes. For example:

```
<finis>::= <skip> 'END' | 'end'
    <skip> ';' ['END'];
```

will bypass blanks, find the keyword "END" in either upper or lower case, bypass any following blanks, and, if a semicolon follows, output the character string "END" (in upper case, as specified).

Numeric output is accomplished by enclosing one or more pairs of hexadecimal digits in angle brackets, as follows:

```
<S370 SVC Op Code>::= [<0A>];
```

Nodal output is accomplished by specification of the subordinate node's position relative to the output specification: where the most recent output-producing node is 1, the next most recent is 2, and so on. In the following example:

```
<pointer>::= <internal label>
    ['DC AL4(' 1 ')'];
```

if "internal label" is satisfied, the specification will output the string "DC AL4(" and follow it with the output of the "internal label" specification, concluding with the closing right parenthesis. When multiple nodal output specifications are used together, they must be separated by commas.

Note that, for node numbering purposes, output specifications are not considered to be *output-producing* specifications. Also, for output purposes, a sequence of specifications separated by vertical bars is considered to be a single node.

Assembler Code Insertion and Built-In Functions

Assembler code may be inserted at any point between specifications by enclosing the code in double angle brackets, as follows:

```
<< Title 'PL/C [V2.0]
    COMPILER MAINLINE' >>
```

An assembler instruction may be inserted at any point between specifications in the containing statement by coding an ampersand (&) followed by the assembler op code mnemonic or macro name. If operands are used, they are enclosed in parentheses and must immediately follow the operation mnemonic. For example:

```
<dummy spec>::=
    &COPY(DSNAME) &CSECT;
```

A minimal set of built-in functions has been implemented for the current PL/C compiler using this facility. For example, a SKIP macro calls an assembler language subroutine to advance the scan pointer to the next nonblank character; this approach speeds up the PL/C compiler considerably. Another example is the MEMB macro, which generates a call to another assembler language subroutine to check the character at the current scan point for membership in the character string supplied as an argument to the macro. Thus:

```
<decimal digit>::=
    '0' | '1' | '2' | '3' | '4' |
    '5' | '6' | '7' | '8' | '9'[1];
```

can be replaced by the less cumbersome and more rapid:

```
<decimal digit>::=
    &MEMB('0123456789')[1];
```

This permits the incorporation of any special facility that may be required either for performance or to meet language requirements (such as attribute analysis for PL/C structure elements). We recommend that this method be reserved for incorporation of new PL/C facilities and that target compiler code be generated using the text inclusion method (angle brackets).

Development History

The PL/C [V1.0] compiler (the initial version) was implemented in Technical Design Labs' Zapple BASIC on a

Z80-based microprocessor system. The entire program required approximately 550 BASIC statements and was adequate only to compile the PL/C compiler itself. The PL/C [V2.0] compiler required only 44 PL/C statements, which was fortunate since the PL/C [V1.0] compiler processed approximately 15 statements per hour.

The output of the BASIC version was a string of macro assembler statements, all of which were macro invocations describing common high-level compiler functions. These macro invocations were generated in a form acceptable to most macro assemblers (as were those of subsequent versions), so the PL/C compiler has existed in a processor-independent form from its earliest stages.

A set of approximately two dozen macros was written to generate Z80 microprocessor instructions. Nearly all of the macros were expanded to compiler subroutine calling sequences, and the appropriate set of compiler subroutines was written in Z80 assembler language. These macros and subroutines are the only processor-dependent code involved in implementing the PL/C compiler.

The common subroutine source code file was concatenated to the generated compiler mainline file, as was the macro definition file. The composite assembler source file was assembled using Technical Design Labs' macro assembler program.

PL/C [V2.0] was up and running after eight iterations of the process just described. This version was functionally equivalent to PL/C [V1.0] and compiled at a rate of approximately two statements per second – a considerable speed improvement over the original BASIC program! At this point, the PL/C source code was rewritten to provide much greater power and flexibility, additional built-in functions were brought into play, and existing functions were made more flexible.

In generating PL/C [V3.0], which produces Z80 BASIC assembler source code, the macro definitions were maintained in a separate file and were brought into play via an '.INSERT' statement generated as part of a prolog generated by PL/C [V2.1]. The common subroutines were maintained in a separate relocatable object library and were not made a part of the object compiler until link-edit time. This approach permitted separation of compiler components into segments that were much easier to maintain.

PL/C [V3.0] was judged adequate for generation of additional compilers; with this version, we considered the PL/C programming language implementation to be complete. Additional PL/C compiler programming activity is categorized as either maintenance or enhancement.

Availability of PL/C

Readers may obtain a personal-use (non-commercial) PL/C license and Z80-executable versions of PL/C Compilers which generate TDL Assembler and Macro code, the PL/C Subroutine Library in relocatable object form, and PL/C User Manual for $25. On receipt of payment I will send a license agreement to be signed and returned. On receipt of the license agreement I will mail the diskette. Commented source listings of the PL/C subroutine library (on diskette) and commercial licenses are also available.

DDJ

(Listings begin on page 50)

Table I.

```
<program>::= <statement>$ <end>;

<statement>::= <generate> | <compiler statement>;

<generate>::= ' '$* '<<' &NEXT('>>');

<compiler statement>::= <statement label> <statement body>;

<statement label>::= <compiler label>  ' '$* ':' ':='*;

<compiler label>::= ' '$* '<' <name root> <name segment>$*
                    ' '$* '>';

<name root>::= ' '$*  <letter> <alphanumeric>$*;

<letter>::= <lower case> | <upper case>;

<lower case>::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' |
                'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' |
                'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
                'y' | 'z';

<upper case>::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' |
```

(Continued on next page)

(Table 1. Continued)

```
                'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' |
                'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
                'Y' | 'Z';
<alphanumeric>::= <letter> | <digit>;
<digit>::= '0' | '1' | '2' | '3' | '4' |
          '5' | '6' | '7' | '8' | '9';
<name segment>::= ' ' ' '$# <alphanumeric>$;
<statement body>::= <action spec> <semi> | <statement body>;
<action spec>::= <generate> | <scan item> | <output item>;
<scan item>::= <scan call> | <macro call> | <scan string> |
              <scan value> | <parenthetic group>;
<scan call>::= <compiler label> <modifier>*;
<macro call>::= ' '$# '&' <built in> | <user macro>;
<user macro>::= <name root> <macro parameter list>*;
<macro parameter list>::= ' '$# '(' <macro arg>
                          <addl macro arg>$# ' '$# ')';
<macro arg>::= <string> | <macro value> | <name root>;
<string>::= ' '$# '''' <string character>$ '''';
<string character>::= <alphanumeric> | <special>;
<special>::= ' ' | '!' | '"' | '#' | '$' | '%' | '&' | '(' |
            ')' | '*' | ':' | '=' | '-' | '{' | '}' | '[' |
            ']' | '~' | '^' | '+' | ';' | '@' | '|' | '\' |
            ' ' | '<' | '>' | ',' | '.' | '?' | '/' | '''''' ;
<macro value>::= ' '$# <digit>$;
<addl macro arg>::= ' '$# ',' <macro arg>;
<scan string>::= <string> <modifier>*;
<scan value>::= ' '$# '#' '''' <hex pair>$ ' '$# ''''
               <modifier>*;
<hex pair>::= ' '$# <hex digit> <hex digit>;
```

```
<hex digit>::= <digit> | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';

<parenthetic group>::= ' '$# '(' <action spec>$ ' '$# ')'
                       <modifier>*;

<modifier>::= ' '$# <optionally many> | <one or more> |
              <optional> <alt>*;

<optionally many>::= '$' ' '$# '*';

<one or more>::= '$';

<optional>::= '*';

<alt>::= ' '$# '|' <scan item>;

<output item>::= ' '$# '[' <output spec>$ ' '$# ']';

<output spec>::= <node spec> | <output string> | <output value>;

<node spec>::= <node number> ( ' '$# ',' <node number> )$#;

<node number>::= ' '$# <digit>$;

<output string>::= <string>;

<output value>::= ' '$# '<' ' '$# <hex pair>$ ' '$# '>';

<semi>::= ' '$# ';';

<end>::= ' '$# 'END' | 'end' <semi>;

end;
```

(Listing begin on page 50)

PL/C Compiler Listing (Text begins on page 44)

Listing One

```
<<.TITLE        'PL/C [2.2] Programming Language for Compilers'
.SBTTL  'Copyright (C) 1981 - MRD Systems, Inc., POB 147, Spring Valley, MN'
.MAIN.::CALL    $INIT#          ; Initialize Compiler
>>
<program>: <statement>$ <end> [2,1];

<statement>: <gen block> | <compiler statement>;

<gen block>: <generate> [1] &FLSH;

<generate>: &SKIP '<<' &COLL(ON) &NEXT('>>') &COLL(OFF) [1];

<compiler statement>: <statement label> <statement body>
        [2.1'   RTRN
'] &FLSH;

<statement label>: <compiler label> &SKIP ':' ':='# &SYMB(3) ['
;  '4'

'1':'];

<compiler label>: &SKIP '<' <label root> <label segment>$# &SKIP '>' [3,2];

<label root>: &SKIP &COLL(ON) <letter> <alphanumeric>$# &COLL(OFF) [2,1];

<letter>: <lower case> | <upper case> [1];

<lower case>: &MEMB('abcdefghijklmnopqrstuvwxyz') [1];

<upper case>: &MEMB('ABCDEFGHIJKLMNOPQRSTUVWXYZ') [1];

<alphanumeric>: <letter> | <digit> [1];

<digit>: &MEMB('0123456789') [1];

<label segment>: ' ' &SKIP &COLL(ON) <alphanumeric>$ &COLL(OFF) [' '1];

<statement body>: <action specification> <semi> | <statement body> [2,1];

<action specification>: <generate> | <scan item> | <output item> [1];

<scan item>: <scan call> | <macro call> | <scan string> | <scan value> |
              <parenthetic group> [1];

<scan call>: <compiler label> <modifier># &SYMB(2)
       ['      XFER    '1','2'         ; '3'
'];

<macro call>: &SKIP '&' <built in> | <user macro> [1];

<built in>: <next> | <member> | <collect> [1];
```

(Continued on page 52)

PL/C Compiler Listing (Listing continued, text begins on page 44)

Listing One

```
<next>: 'NEXT' | 'next' &SKIP '(' <string> &SKIP ')' <modifier>*
        ['      NEXT    '3','1'
'];

<member>: 'MEMB' | 'memb' &SKIP '(' <string> &SKIP ')' <modifier>*
        ['      MEMB    '3','1'
'];

<collect>: 'COLL' | 'coll' &SKIP '(' <col on> | <col off> &SKIP ')'
        ['      COLL    '2'
'];

<col on>: 'ON' | 'on' ['ON'];

<col off>: 'OFF' | 'off' ['OFF'];

<user macro>: <label root> <macro parameter list>* ['   '2,1'
'];

<macro parameter list>: &SKIP '(' <macro arg> <addl macro arg>$* &SKIP ')'
        ['      '3,2];

<macro arg>: <string> | <macro value> | <label root> [1];

<string>: &SKIP '''' &COLL(ON) <string character>$ &COLL(OFF) '''' [<60>2<60>];

<string character>: <alphanumeric> | <special> | <string quote> [1];

<special>: &MEMB(' !"#$%&()*:=-{[]}~^+;@|\ <,>.?/
        ') [1];

<string quote>: &COLL(OFF) '''''' &COLL(ON) [''''];

<macro value>: &SKIP &COLL(ON) <digit>$ &COLL(OFF) [1];

<addl macro arg>: &SKIP '.' <macro arg> ['.'1];

<scan string>: <string> <modifier>* ['  SCAN    '2','1'
'];

<scan value>: &SKIP '#' '''' <hex pair>$ &SKIP '''' <modifier>*
        ['      SCNR    '<60>3<60>'.'1'
'];


<hex pair>: &SKIP &COLL(ON) <hex digit> <hex digit> &COLL(OFF) [2,1];

<hex digit>: &MEMB('0123456789ABCDEF') [1];

<parenthetic group>: &SKIP '(' <action specification>$ &SKIP ')' <modifier>*
        &LGEN &LGEN ['  XFER    '2','3'
        GOTO    '1'
'2':'5' RTRN
'1':'];
```

```
<parenthetic body>: <action specification> &SKIP ')' | <parenthetic body> [3,1];

<modifier>: &SKIP <alt mod> | <optionally many> | <one or more> |
        <optional> | <default zero> [1];

<alt mod>: <multi alt> | <single alt> [1];

<multi alt>: '$' &SKIP '|' ['6'];

<single alt>: '|' ['4'];

<optionally many>: '$' &SKIP '*' ['3'];

<one or more>: '$' ['2'];

<optional>: '*' ['1'];

<default zero>: ['0'];

<output item>: &SKIP '[' <output spec>$ &SKIP ']' [2];

<output spec>: <node spec> | <output string> | <output value> [1];

<node spec>: <node number> <addl node number>$* [2,1];

<node number>: &SKIP &COLL(ON) <digit>$ &COLL(OFF)
        ['      NODE    '1'
'];

<addl node number>: &SKIP ',' <node number> [1];

<output string>: <string>
        ['      STOP    '1'
'];

<output value>: &SKIP '<' &SKIP <hex pair>$ &SKIP '>'
        ['      OHEX    '''2'''
'];

<semi>: &SKIP ';';

<end>: &SKIP 'END' | 'end' <semi>
        ['      .END
'<1A>];

end;
```

End Listing One

Listing Two

```
.TITLE  'PL/C [2.2] Programming Language for Compilers'
.SBTTL  'Copyright (C) 1981 - MRD Systems, Inc., POB 147, Spring Valley, MN'
.MAIN.::CALL    $INIT#          ; Initialize Compiler

;  program

X0001:  XFER    X0002,2         ; statement
        XFER    X0003.0         ; end
```

(Continued on next page)

PL/C Compiler Listing

Listing Two (Listing continued, text begins on page 44)

```
        NODE    2
        NODE    1
        RTRN

;  statement

X0002:  XFER    X0004,4         ; gen block
        XFER    X0005,0         ; compiler statement
        RTRN

;  gen block

X0004:  XFER    X0006,0         ; generate
        NODE    1
        FLSH
        RTRN

;  generate

X0006:  SKIP
        SCAN    `<<`,0
        COLL    ON
        NEXT    `>>`,0
        COLL    OFF
        NODE    1
        RTRN

;  compiler statement

X0005:  XFER    X0007,0         ; statement label
        XFER    X0008,0         ; statement body
        NODE    2
        NODE    1
        STOP    `       RTRN
`
        FLSH
        RTRN

;  statement label

X0007:  XFER    X0009,0         ; compiler label
        SKIP
        SCAN    `:`,0
        SCAN    `:=`,1
        SYMB    3
        STOP    `
;  `
        NODE    4
        STOP    `
`
        NODE    1
        STOP    `:`
        RTRN
```

```
;  compiler label

X0009:  SKIP
        SCAN    `<`,0
        XFER    X000A,0         ; label root
        XFER    X000B,3         ; label segment
        SKIP
        SCAN    `>`.0
        NODE    3
        NODE    2
        RTRN

;  label root

X000A:  SKIP
        COLL    ON
        XFER    X000C,0         ; letter
        XFER    X000D,3         ; alphanumeric
        COLL    OFF
        NODE    2
        NODE    1
        RTRN
;  letter

X000C:  XFER    X000E,4         ; lower case
        XFER    X000F,0         ; upper case
        NODE    1
        RTRN

;  lower case

X000E:  MEMB    `abcdefghijklmnopqrstuvwxyz`,0
        NODE    1
        RTRN

;  upper case

X000F:  MEMB    `ABCDEFGHIJKLMNOPQRSTUVWXYZ`,0
        NODE    1
        RTRN

;  alphanumeric

X000D:  XFER    X000C,4         ; letter
        XFER    X0010.0         ; digit
        NODE    1
        RTRN

;  digit

X0010:  MEMB    `0123456789`.0
        NODE    1
        RTRN

;  label segment
```

(Continued on next page)

PL/C Compiler Listing

Listing Two (Listing continued, text begins on page 44)

```
X000B:  SCAN    ` `.0
        SKIP
        COLL    ON
        XFER    X000D,2         ; alphanumeric
        COLL    OFF
        STOP    ` `
        NODE    1
        RTRN

;  statement body

X0008:  XFER    X0011.0         ; action specification
        XFER    X0012.4         ; semi
        XFER    X0008,0         ; statement body
        NODE    2
        NODE    1
        RTRN

;  action specification

X0011:  XFER    X0006,4         ; generate
        XFER    X0013.4         ; scan item
        XFER    X0014,0         ; output item
        NODE    1
        RTRN

;  scan item

X0013:  XFER    X0015,4         ; scan call
        XFER    X0016,4         ; macro call
        XFER    X0017,4         ; scan string
        XFER    X0018,4         ; scan value
        XFER    X0019,0         ; parenthetic group
        NODE    1
        RTRN

;  scan call

X0015:  XFER    X0009,0         ; compiler label
        XFER    X001A,1         ; modifier
        SYMB    2
        STOP    `       XFER    `
        NODE    1
        STOP    `.`
        NODE    2
        STOP    `               ; `
        NODE    3
        STOP    `

        RTRN

;  macro call
X0016:  SKIP
        SCAN    `&`.0
```

```
        XFER    X001B,4         ; built in
        XFER    X001C,0         ; user macro
        NODE    1
        RTRN

;  built in

X001B:  XFER    X001D,4         ; next
        XFER    X001E,4         ; member
        XFER    X001F,0         ; collect
        NODE    1
        RTRN

;  next

X001D:  SCAN    `NEXT`,4
        SCAN    `next`.0
        SKIP
        SCAN    `(`.0
        XFER    X0020.0         ; string
        SKIP
        SCAN    `)`.0
        XFER    X001A,1         ; modifier
        STOP    `       NEXT    `
        NODE    3
        STOP    `.`
        NODE    1
        STOP    `
`
        RTRN

;  member

X001E:  SCAN    `MEMB`,4
        SCAN    `memb`.0
        SKIP
        SCAN    `(`.0
        XFER    X0020.0         ; string
        SKIP
        SCAN    `)`.0
        XFER    X001A,1         ; modifier
        STOP    `       MEMB    `
        NODE    3
        STOP    `.`
        NODE    1
        STOP    `
`
        RTRN

;  collect

X001F:  SCAN    `COLL`.4
        SCAN    `coll`.0
        SKIP
        SCAN    `(`.0
        XFER    X0021,4         ; col on
        XFER    X0022.0         ; col off
```

(Continued on page 62)

PL/C Compiler Listing (Listing continued, text begins on page 44)

Listing Two

```
        SKIP
        SCAN    `)`,0
        STOP    `       COLL    `
        NODE    2
        STOP    `
`
        RTRN

;  col on

X0021:  SCAN    `ON`.4
        SCAN    `on`,0
        STOP    `ON`
        RTRN

;  col off

X0022:  SCAN    `OFF`.4
        SCAN    `off`.0
        STOP    `OFF`
        RTRN
;  user macro

X001C:  XFER    X000A,0         ; label root
        XFER    X0023,1         ; macro parameter list
        STOP    `       `
        NODE    2
        NODE    1
        STOP    `
`
        RTRN

;  macro parameter list

X0023:  SKIP
        SCAN    `(`.0
        XFER    X0024,0         ; macro arg
        XFER    X0025.3         ; addl macro arg
        SKIP
        SCAN    `)`.0
        STOP    `       `
        NODE    3
        NODE    2
        RTRN

;  macro arg

X0024:  XFER    X0020.4         ; string
        XFER    X0026,4         ; macro value
        XFER    X000A,0         ; label root
```

```
        NODE    1
        RTRN

;  string

X0020:  SKIP
        SCAN    `'`.0
        COLL    ON
        XFER    X0027,2         ; string character
        COLL    OFF
        SCAN    `'`,0
        OHEX    '60'
        NODE    2
        OHEX    '60'
        RTRN

;  string character


X0027:  XFER    X000D,4         ; alphanumeric
        XFER    X0028,4         ; special
        XFER    X0029,0         ; string quote
        NODE    1
        RTRN

;  special

X0028:  MEMB    ` !"#$%&()*:=-{[]}~^+;@|\ <,>.?/
        `.0
        NODE    1
        RTRN

;  string quote

X0029:  COLL    OFF
        SCAN    `''`.0
        COLL    ON
        STOP    `'`
        RTRN

;  macro value

X0026:  SKIP
        COLL    ON
        XFER    X0010.2         ; digit
        COLL    OFF
        NODE    1
        RTRN

;  addl macro arg
```

(Continued on next page)

PL/C Compiler Listing

Listing Two

(Listing continued, text begins on page 44)

```
X0025:  SKIP
        SCAN    `.`.0
        XFER    X0024.0         ; macro arg
        STOP    `.`
        NODE    1
        RTRN

;  scan string

X0017:  XFER    X0020.0         ; string
        XFER    X001A,1         ; modifier
        STOP    `       SCAN    `
        NODE    2
        STOP    `.`
        NODE    1
        STOP    `
`
        RTRN

;  scan value

X0018:  SKIP
        SCAN    `#`.0
        SCAN    `'`.0
        XFER    X002A,2         ; hex pair
        SKIP
        SCAN    `'`.0
        XFER    X001A,1         ; modifier
        STOP    `       SCNR    `
        OHEX    '60'
        NODE    3
        OHEX    '60'
        STOP    `,`
        NODE    1
        STOP    `
`
        RTRN

;  hex pair

X002A:  SKIP
        COLL    ON
        XFER    X002B,0         ; hex digit
        XFER    X002B,0         ; hex digit
        COLL    OFF
        NODE    2
        NODE    1
        RTRN

;  hex digit
```

```
X002B:  MEMB    `0123456789ABCDEF`,0
        NODE    1
        RTRN

;  parenthetic group

X0019:  SKIP
        SCAN    `(`,0
        XFER    X0011,2         ; action specification
        SKIP
        SCAN    `)`,0
        XFER    X001A,1         ; modifier
        LGEN
        LGEN
        STOP    `       XFER    `
        NODE    2
        STOP    `.`
        NODE    3
        STOP    `
        GOTO    `
        NODE    1
        STOP    `
`
        NODE    2
        STOP    `:`
        NODE    5
        STOP    `       RTRN
`
        NODE    1
        STOP    `:`
        RTRN

;  parenthetic body


X002C:  XFER    X0011,0         ; action specification
        SKIP
        SCAN    `)`,4
        XFER    X002C,0         ; parenthetic body
        NODE    3
        NODE    1
        RTRN

;  modifier


X001A:  SKIP
        XFER    X002D,4         ; alt mod
        XFER    X002E,4         ; optionally many
        XFER    X002F,4         ; one or more
        XFER    X0030,4         ; optional
        XFER    X0031,0         ; default zero
        NODE    1
        RTRN
```

(Continued on next page)

PL/C Compiler Listing

Listing Two

(Listing continued, text begins on page 44)

```
;  alt mod

X002D:  XFER    X0032.4         ; multi alt
        XFER    X0033,0         ; single alt
        NODE    1
        RTRN

;  multi alt

X0032:  SCAN    `$`.0
        SKIP
        SCAN    `|`.0
        STOP    `6`
        RTRN

;  single alt

X0033:  SCAN    `|`,0
        STOP    `4`
        RTRN

;  optionally many

X002E:  SCAN    `$`,0
        SKIP
        SCAN    `*`.0
        STOP    `3`
        RTRN

;  one or more

X002F:  SCAN    `$`,0
        STOP    `2`
        RTRN

;  optional

X0030:  SCAN    `*`.0
        STOP    `1`
        RTRN

;  default zero

X0031:  STOP    `0`
        RTRN

;  output item

X0014:  SKIP
        SCAN    `[`.0
        XFER    X0034.2         ; output spec
```

```
        SKIP
        SCAN    `]`.0
        NODE    2
        RTRN

;  output spec
X0034:  XFER    X0035.4         ; node spec
        XFER    X0036,4         ; output string
        XFER    X0037,0         ; output value
        NODE    1
        RTRN

;  node spec


X0035:  XFER    X0038,0         ; node number
        XFER    X0039,3         ; addl node number
        NODE    2
        NODE    1
        RTRN

;  node number


X0038:  SKIP
        COLL    ON
        XFER    X0010.2         ; digit
        COLL    OFF
        STOP    `       NODE    `
        NODE    1
        STOP    `
`
        RTRN


;  addl node number

X0039:  SKIP
        SCAN    `.`,0
        XFER    X0038,0         ; node number
        NODE    1
        RTRN

;  output string


X0036:  XFER    X0020.0         ; string
        STOP    `       STOP    `
        NODE    1
        STOP    `
`
        RTRN

;  output value
```

(Continued on next page)

PL/C Compiler Listing

Listing Two

(Listing continued, text begins on page 44)

```
X0037:  SKIP
        SCAN    `<`.0
        SKIP
        XFER    X002A,2          ; hex pair
        SKIP
        SCAN    `>`.0
        STOP    `        OHEX    '`
        NODE    2
        STOP    `'
`
        RTRN

;  semi


X0012:  SKIP
        SCAN    `;`,0
        RTRN

;  end


X0003:  SKIP
        SCAN    `END`,4
        SCAN    `end`,0
        XFER    X0012.0          ; semi
        STOP    `        .END

        OHEX    '1A'
        RTRN
        .END
```

End Listing

"Today, class, I would like to explain the advantages of the hexadecimal system."

Program Design Using Pseudocode

by Ken Takara

Ken Takara, 1306 Sippola Way, San Jose, CA 95121.

"Pseudocode is a sort of 'fake' programming language, one that is meant to give you lots of flexibility in designing a program."

Programming involves two distinct steps: design and coding. Programmers are often quite adept at coding but deplorably inadequate when it comes to working out the design part.

When you code, your main concern is the language and all the details of program construction: how to pass parameters to the function, how to set up the search loop, or whether the loop condition is checked at its beginning or end. It's something like taking a trip. Which road should we take? When should we stop for gas? What if the Disneyland Hotel has no vacancies?

Designing is analogous to planning the trip. Where shall we go? What shall we do? Shall we drive or fly? Unfortunately, planning the trip, like wrestling with the design of a program, is hardly ever as exciting as actually *doing* it. It must be human nature to say, "Aw, heck, I'll figure it out as I go." The result is usually pretty chaotic. "Let's see, was INREC 8 bytes or 10 bytes long? I assumed 8 here, but I used 12 over there . . . . "

Except for a really trivial program, it is unreasonable to assume that you can design while coding or code while designing. Usually, the design loses out since you tend to concentrate more on coding detail (the *how* part) than on design detail (the *what it does* part).

Numerous tools are available for designing programs. Flow charts are probably the most heavily abused of them. Some structured design methodologies, complete with a plethora of rigid rules for their usage, are also available. Then there is pseudo-code.

Pseudo-code is a sort of "fake" programming language, one that is meant to give you lots of flexibility in designing a program. At its worst, pseudo-code can be so restrictive that it becomes much like another programming language. At its best, it allows you to create structures that make sense only to you, although they may be meaningless to a compiler.

In this article, I will present a way of using pseudo-code during the design process. Although I don't intend to set down any coding conventions, I will describe the conventions I used at the end of the article. Our concern here is the method of usage, not the method of coding.

Design Philosophy

We are going to design a game program. We will start by describing it in very general, rather vague terms. We will then build a modular design using structured English, working from the top down. We will ignore such things as subroutines, variables (for the most part), structures, and primitives. It won't look anything like a program.

Then we will get into pseudo-code. We will translate each module into a routine using pseudo-code, creating more routines as we go. We will also start creating variables, both local and global, and we will reference several data structures. Because of the inexact nature of the design process, we will probably reiterate the various stages several times, reorganizing modules and rebuilding data structures. After that, we'll think about coding.

A Shoot-'Em-Up Game

You are playing a video game. In the center of the screen is the cross hair, against a background of fixed stars. As you move the joystick around, the background moves also. It's as if you were in a spaceship, in the gunner's compartment, scanning nearby space with your missile launcher. A series of fighter craft comes into view! They weave about against the fixed background, and the whole scene moves about the screen as you maneuver the joystick to take aim. When you press the fire button, a missile is launched. Each time a missile intercepts an attacking fighter, you see an explosion and are awarded points. You play until a certain number of ships gets past you, at which point the game ends. The number of fighters on the screen during a "wave" may be limited, as well as the number of missiles on the screen at a time.

Does it sound exciting? No? Not a video game addict, eh? Well, bear with me. The principles we will use to design this game are applicable elsewhere, too. Let's now do what I call a Level 1 design specification, using structured English. We may have to rework the design several times, but we should have a decent one by the time we're done.

Level 1 Design

Designing usually requires lots of paper, most of which ends up in the trash. We'll skip the scribbling and go straight to the final draft. Our top down design begins with the main module.

Main Module

1. Create attacking fighters and set their initial positions.
2. Set initial coordinates of cross hairs against the reference background.
3. Draw the initial screen.
4. Until the number of fighters that "escape" is greater than the limit, do the following:
   a. See if any missiles are fired (Fire Control).
   b. Check for interceptions (Intercept Control).
   c. Move the fighters (Fighter Movement).
   d. Move any missiles (Missile Control).
   e. Determine cross hair position (Scan Control).
   f. Draw new screen (Display Module).
   g. Delay for skill level.
   h. Loop back to 4.
5. End of game; display final tally.

Fighter Movement

Given that we know the previous x,y position of each fighter and its range from us:

For each fighter, calculate a new x,y position against the reference background and a new range.

Missile Control

Given that we know the previous

range of the missile from the player (and assuming that its x,y reference coordinates don't change):

Calculate its new range.

Scan Control

Given that we know the previous x,y position of the cross hairs against the reference background:

1. Read the x,y values of the joystick and convert to (±1, ±1) values.
2. Change the x,y values by adding the results from the joystick. This implies that moving the joystick causes the center of the screen to drift against or "scan" the reference background.

Intercept Control

1. For each missile:
   a. Get its x,y position and range.
   b. Calculate the range it will have after its next missile movement, and call it range 1.
2. For each fighter whose x,y position is the same as that of the missile, if missile range $\geqslant$ fighter range $\geqslant$ range 1:
   a. Get the point value of the fighter.
   b. Eliminate the missile and fighter.
   c. Mark an explosion at the fighter's location.
3. Return the total score.

We'll spare you the rest of the modules; suffice to say that Fire Control checks the fire button, and Display Module displays the screen, showing fighters, missiles, explosions, and background stars.

There are quite a few implications and assumptions here. We imply that a data structure exists for each of the fighters, missiles, and explosions. We imply variables for keeping the score and constants to regulate the game speed and number of rounds to play. We assume that we can read the joystick and convert its values and that we can read the fire button.

It's a good idea to sit back and reconsider the game in its entirety just so that we don't become too myopic from staring at the details. For example, notice that Intercept Control checks a missile's next position. The Missile Control module also needs this information in order to move the missile. We might want to calculate the position once then save it for use by other modules. We should make a note of this and modify the design to take this into account. If we noticed this while coding, major surgery might be required on our beautifully written program to accommodate this discovery.

Notice also that, despite all these assumptions and implications, we have not specified how any of this will be done. We simply assume that it can be done. As programmers, we ought to know if it can be done or not, and we ought to know the limits of the machine. In any event, we have specified what it does without worrying about how it will do it.

At this point, we can easily change our minds concerning the design of the game. We can let the fighters shoot back, or we can describe various classes of fighters with different abilities. We have this flexibility because we have yet to set up the rigid data and control structures.

Once we are satisfied with our base idea, we go on to something closer to a program. We can start using data structures and variables and flow-of-control, and leave the somewhat ambiguous structured English of Level 1. Figure 4(a) (page 75) presents the conventions adopted for Level 1 design for use as guidelines in subsequent design efforts.

Level 2 Design

During Level 1, we described a set of modules in rather loose terms, implying all sorts of data and control structures, as well as numerous functions and subroutines, without really acknowledging them as such. In Level 2, we transform the modules into routines using real, live pseudo-code. Although we use variables, data structures, and certain control structures, we aren't really coding in the normal sense. We are still describing what the routines do but in sufficient detail to make the how of it easy, even obvious, to work out.

We'll start by identifying what might be global variables. These are data structures and other variables that seem to be used in different modules, retaining the changes that other modules make to them. For example, the fighters might be described by a collection of data structures called SHIPREC, while missiles are described by a collection of data structures called MISSILE. Some obvious variables are the total number of points, current center-of-screen coordinates, and the number of fighters that have "escaped."

For Level 2 designs for the Main Module and the Intercept Control routines, we'll again skip the scratchpaper stage and present a (nearly) finished draft. Figure 1 (page 72) shows the logic for the Main Module.

The Main Module routine uses a number of routines, most of which are familiar from Level 1. The notation SHIPREC (i) does not imply that SHIPREC is an array; it is simply a convenient fiction to indicate "this particular SHIPREC" as opposed to "any SHIPREC." We have described only those variables and constants actually used by this routine, although many other global variables exist; we will indicate global variables in other routines as we come to them to keep our list of variables and constants from getting out of hand. We didn't need to describe each of the routines derived from the modules, but we did it for completeness.

We also used two loop structures, one of which looks vaguely like the "do until" loop. The other is easy to understand but looks unlike any "real" loop structure in any language. The two loop structures are actually pseudo-loops, in that they aren't supposed to tell you how to code the loop or even which loop to use. They only tell you what to loop on and when to quit the loop. More on this later.

Of course, the top level module of many programs can be awfully dull; nothing really happens there. All the action is found in the lower level routines. Figure 2 (page 74) shows the pseudo-code for the Intercept Control routine.

The MX, MY, MR, and MR1 variables are local and temporary. Depending upon the language, they may be represented by registers, values on stacks, etc. In some cases, they may never be really created.

The functions, XPOSITION.M, YPOSITION.M, XPOSITION.S, and so on may be fictitious, depending upon the actual format of the fighter records and missile records. They are created here for convenience. The explicit difference between XPOSITION.M and XPOSITION.S, and between YPOSITION.M and YPOSITION.S, and so on, is because we don't know anything about the actual layouts of the missile and fighter records.

The routine NEWSHIP has been re-used, indicating that it may be global or at least defined at a higher level than this routine. When designing and coding it, we will have to keep in mind that it can be called at any time, not only during the initial setup.

Although we have identified some of the fields within the fighter and missile records, we still do not impose a format on them. We may find that we need more fields in some of these records. For these reasons, we reference the fields within the records by name, using the "dot" convention – RECORD.FIELD1 refers to a field called FIELD1 in a data structure called RECORD.

In both of the routines that we pseudo-coded, we referred to the fighter and missile records with miniscule subscripts. This is not intended to imply that they are arrays. We have not specified what their actual structure is and simply use the miniscule subscript to single out a particular record and to imply an ordering to the records. That is, some "first" record exists and for any record a "next" record exists. These records could be arrays, or they could be elements in a linked list or sequential blocks in memory. The physical layout is not important at this time.

Figure 4(b) (page 75) lists the conven-

Main Module Logic

```
MAIN.MODULE: routine;
    for each SHIPREC(i) do                    (init fighter records)
        call NEWSHIP (SHIPREC (i));           (create ship record)
    loop (i);                                 (next rec)
    FAILURE.COUNT := 0;                       (failure counter)
    call DISPLAY.SCREEN;                      (draw 1st screen)
    until FAILURE.COUNT=MAXFAIL do            (main prog loop)
        call FIRE.CONTROL;                    (launch missiles)
        SCORE := SCORE + INTERCEPT;           (Intercept Ctl)
        call FIGHTER.MOVEMENT;                (move fighters)
        call MISSILE.CONTROL;                 (move missiles)
        call SCAN.CONTROL;                    (new x,y cross hairs)
        call DISPLAY.SCREEN;                  (draw new screen)
        call SPEED.DELAY;                     (slow game)
    again;
    call END.GAME;                            (final display)
end MAIN.MODULE;
```

Variables and Constants

SHIPREC
Global structure giving positions of fighters (x,y, range) as well as other information.

SCORE
Global variable with current score.

FAILURE.COUNT
Variable giving number of ships that have "escaped."

MAXFAIL
Constant giving maximum number of failures before game ends. Suggested value=10.

Functions and Routines

NEWSHIP (ship record)
Initializes a record, creating a new fighter, position, value, etc.

DISPLAY.SCREEN
Draws a screen with current objects (fighters, missiles, explosions, stars, etc.).

FIRE.CONTROL
Performs missile launches.

INTERCEPT
Returns total point value of any fighters shot down by missiles. Checks for interception and eliminates hit fighters and missiles. Records explosions.

FIGHTER.MOVEMENT
Moves each of the fighters.

MISSILE.CONTROL
Moves each of the missiles.

SCAN.CONTROL
Checks joystick and sets new center-of-screen coordinates for cross hairs.

SPEED.DELAY
Slows down the action by inserting a delay.

END.GAME
Performs end-of-game cleanup and display.

Figure 1.

tions used for Level 2 design.

Loops and Other Control Structures

A dozen different loop constructs no doubt are available among programming languages, as well as a dozen ways to use them with varying degrees of effectiveness. If you use an "until" loop, does it check the condition at the beginning of the loop or at the end? What is the value of the loop counter after the loop terminates? Is the loop always executed at least once?

Many of the loop peculiarities depend upon the implementation. We would like to have a set of pseudo-loops that always does what we want it to do. Since we are working with pseudo-code, there is no reason why we can't create such loops. Let's assume that our loops will be executed zero times if the exit condition is met immediately upon entry to the loop. Let's also assume that loop counters retain their values after the loop is exited. We know that we can ensure these two points during real coding, even if the language doesn't support them, by adding our own checks or by using different loop constructs.

Three things are common to loops. First, they assume some initial condition or initial item: the initial value of a counter, for example, or the first element in a linked list. Second, most loops assume that a "next" item follows the first. Third, the program can exit the loop either when the elements are exhausted or upon realization of an "exit" condition of some sort. Regardless of the peculiarities of any language, loop structures almost always have provision for these three points. Of course, we may have to do some twisting to use them the way we want to, but that is why we work out the design in pseudo-code to begin with.

Given that we can have our pseudo-loops do what we want them to do, what sort of loop constructs might we use? We suggest the following types:

1. The "until X do . . . again" loop. If condition X exists upon entry to the loop, the loop is not executed.
2. "while X do . . . again" is similar to (1) but runs while condition X holds. If X is not true on entry, the loop is not executed.
3. "for each X do . . . loop" or "for each X (i) (for i=m to n) do . . . loop" runs the loop until the elements X are exhausted.
4. "do . . . if X then exit . . . again" exits the loop upon satisfaction of the test.

Two other control structures that are frequently used are the "if . . then . .

Intercept Routine Logic

```
INTERCEPT: Function;
POINTVAL := 0;                          (initialize point count)
for each (MISSILE(i)) do                (another loop)
    MX := MISSILE(i).XPOS;              (current x coordinate)
    MY := MISSILE(i).YPOS;              (current y coordinate)
    MR := MISSILE(i).RANGE;             (range from player)
    MR1 := MR + MISSILE(i).SPEED        (find next position)
    for each (SHIPREC(j)) do            (check each target ship)
        if SHIPREC(j).XPOS=MX           (is fighter at the same . . . )
        and SHIPREC(j).YPOS=MY          (position as missile?)
        then
            if MR≤SHIPREC(j).RANGE≤MR1 then          (a hit!)
                POINTVAL := POINTVAL + SHIPREC(j).VALUE
                call ELIMINATE(MISSILE(i));
                call NEWSHIP (SHIPREC(j));
                exitloop(j);            (exit SHIPREC loop)
            endif
        endif;
    next(j);
next(i);
return (POINTVAL);
```

Variables and Constants

POINTVAL
Sum of points added this turn.

MX, MY, MR, MR1
Holding variables for missile position (x,y,range) and projected range of missile.

SHIPREC
Global data structure describing fighters. Includes position and value information. See description below.

MISSILE
Global structure for missiles. See description below.

MISSILE and SHIPREC Structures

MISSILE(record).XPOS
X position of a missile.

MISSILE(record).YPOS
Y position of a missile.

MISSILE(record).RANGE
Distance of a missile from the player.

MISSILE(record).SPEED
Velocity at which a missile is traveling.

SHIPREC(record).XPOS
X position of an attacking fighter craft.

SHIPREC(record).YPOS
Y position of a fighter.

SHIPREC(record).RANGE
Distance from player to attacking fighter.

SHIPREC(record).VALUE
Point value of a target fighter.

SHIPREC(record).

Functions and Subroutines

ELIMINATE(missile record)
Deletes a missile record (either by unlinking it, if a linked list is used, or by setting an "unused" flag).

NEWSHIP(ship record)
Replaces a destroyed fighter with a new one.

NEWBANG(x,y,range)
Makes an entry in the explosion table for the display.

Figure 2.

else . . endif" and the "case . . of . . endcase" structures. Everyone knows the first structure. We like to close each "if" with an "endif." You may or may not, depending on how you feel. The "case . . of . . endcase" structure may be thought of as a selection of one out of several subroutines. For example:

```
case (I) of
    (1) DOTHIS;
    (2) DOTHAT;
    else DOTHEOTHER;
endcase
```

performs the routine DOTHIS if the selector, I, is 1; it does DOTHAT if I=2; and it performs DOTHEOTHER for any other value of I. It is easy to implement even in languages lacking the "case" structure.

The control structures presented here should be sufficient to make the intended structure of any program clear. Often their actual implementations during coding become obscure due to language peculiarities, but a quick reference to the pseudo-code version should serve to remind you of the intended structure. Thus, you will not lose the design of your program while you are struggling with the problems of coding.

Routines and Functions

We started the design process with a loose description of what we wanted to do. We then formalized it somewhat during Level 1 by breaking the process into modules, written in structured English. From there we went on to Level 2, where we transformed the structured English to pseudo-code. At each step we assumed and used numerous subroutines and functions. So where do we describe the subroutines and functions?

There are three classes of routines. One class of routines includes those described as Level 1 modules; these, of course, we translated to Level 2 routines. We briefly described the next class of routines in the "Functions and Subroutines" part of the figures, some of them with sufficient detail to go on to pseudo-coding them. The last set are those routines that are sufficiently complex to require whole Level 1 descriptions of their own. For example, a sophisticated parsing routine within another program might be sufficiently complicated to justify having its own thorough design.

By this time, you are probably pretty familiar with your design, maybe even a little sick of it. It might be reasonable to try coding a small test program to see how it looks. You may discover something that you hadn't anticipated that impacts on your design. For example, the method you selected to draw the display might prove to be far too slow. You experiment

to find a faster way, but it involves a major design change. Oh well, it's better to know now before you have logged a hundred hours of coding and debugging time.

In designing a program, you should defer "programmer" problems until the final stages. But before you start the last design step, it definitely helps to consult the computer. What you learn by building a small test system will help as you struggle with the Level 3 design. Machine dependencies have a way of wrecking "machine-independent" designs just as easily as ignorance of the design can mangle your tidy code. As designer and programmer, you catch it from both sides.

Level 3 Design

We have now pseudo-coded most of the routines in our program. We know what the modules are and what they are to do. We also know about a whole mass of variables and data structures. Now we need to know what the data structures look like and how they are accessed. At Level 3, we design the data structures and the primitive operations on them.

At this point, we become concerned with the selection of the language, as well as language and machine dependencies. We should restrict ourselves, however, to dependencies relative to data structures. We will still use the techniques of Level 2 for specifying control structures and avoid implementation dependencies for these as yet; although we must address the dependencies regarding data structures, we are still involved with design.

For this example, let's pseudo-code the NEWSHIP routine. We must also specify the layout of the fighter record SHIPREC with respect to some language. Let's assume that the language we are using permits us to reference the record and its fields by specifying a name and and offset. We first describe SHIPREC in Figure 3 (page 74).

We have specified, for purposes not connected with the NEWSHIP routine, to include a "next record" pointer in the fighter record, implying that the fighter records constitute a linked list. We have also referenced a random number generator, which we hope is at least partially supplied by the language we intend to use.

Eventually, we will have all the routines pseudo-coded in a similar manner. Figure 4 (c) (page 75) presents the Level 3 design conventions. Assuming that we are still satisfied with our design, we can go on to the final step: coding.

From Pseudo-Code to Real Code

Once we decide to start coding, we should put all our energy to writing code, trusting that our design is complete. The actual process of coding, of course, depends upon the language we have selected. However, we might follow certain general procedures. When we embarked upon design, we worked from the top down. When coding, we work from the bottom up except for the variables, which usually must be declared prior to their use.

Generally, the transformation from pseudo-code to real code is pretty painless, for an excellent reason: you have made all the global decisions and can now concentrate on the local, line-by-line ones of coding! The loop structures, however, always seem to have it in for us. Somehow, loops always do something other than what we want them to do, forcing us to find some way of getting them to do our bidding.

As we mentioned before, most loops imply the existence of an initial element, a successor element, a final element, and some exit method. In coding a loop, then, we should find the most effective way of specifying these points. For example, in our pseudo-code we might have said, "for each SHIPREC(i) do . . " Instead of "for I=1 to n" this might translate to:

```
PTR.S := FIRST.SHIPREC;
until PTR.S=0 do . . .
PTR.S := PTR.S+0
again
```

Summary

The design process begins with an informal description of the program in terms of its actions. This is separated into modules, each of which is stated in structured English. The focus is on what the modules do, and on which one does what, rather

Structure

SHIPREC

offset	name	size	description
+0	NEXTREC	2 byte	Pointer to next record (linked list)
+2	XPOS	2 byte	X coordinate of record
+4	YPOS	2 byte	Y coordinate of record
+6	RANGE	2 byte	Distance from player
+8	VALUE	2 byte	Point value of ship

New Fighter Routine

```
NEWSHIP(SHIP) : Routine;
SHIP.XPOS := RANDOM(0,255);                    (random X coordinate)
SHIP.YPOS := RANDOM(0,255);                    (random Y coordinate)
SHIP.RANGE := RANDOM(MAXRANGE*3/2,MAXRANGE);
                                               (random distance)
SHIP.VALUE := 100*RANDOM(1,3);                 (random value)
end NEWSHIP;
```

Variables and Constants

SHIP
Local reference to a SHIPREC record.

MAXRANGE
Constant giving maximum start distance for attacking fighter.

Functions and Routines

RANDOM(m,n)
Returns a random number between m and n.

Figure 3.

than on how it is done. During this Level 1 design process, data structures are implied by the actions being performed and are not explicitly described.

During Level 2 design, the tight English modules are converted into routines and functions composed of calls to other routines, held together by control structures. The control structures mirror the various structures available in most languages, but they are assumed to do what we want them to do rather than what the language designer specified they should do. Loops check conditions upon entry and may be skipped altogether if the exit conditions are already in effect. The data structures and variables implied during Level 1 are stated, but their actual formats are left vague. Data types are kept simple; integers, real numbers, pointers, characters, and strings are commonly used. You should feel free, however, to create any new data types you want.

Level 3 design starts by describing the data structures and variables used during Level 2 design. Language and machine dependencies should be taken into account at this point. Primitives that work on the data structures are pseudo-coded, still describing what they do, except that now they are designed with the language and machine in mind. We may also wish to annotate routines created during Level 2 design if they happen to directly manipulate any of the data structures.

Finally, the pseudo-coded routines are translated to the actual program code. The various routines are laid out as skeletal forms, and the variable declarations are coded. The routines are coded starting with the primitives, working up to the higher level routines. Routines should be tested as they are coded to simplify the problems associated with integration.

The transformation of loop structures, "if . . then . . " structures, and "case . . " structures should be watched, as language peculiarities can cause difficulties.

If you use this method conscientiously, you should be able to work out the design of a program before you get entangled in the details of coding. Coding itself should be much easier since you can ignore design considerations at that point. Working from Level 2 and Level 3 pseudo-code, even if you are working with the most primitive and unstructured of languages, you should be able to see the program structure easily by comparing the real code with the pseudo-code. Program documentation should be easier, too, as the pseudo-code provides the basis for program comments.

■■J

These are the conventions used for the various design levels in this article. They should be taken as guidelines for working out the design and are not meant to be strictly adhered to.

(a) Level 1 Conventions

Modules are worked out from the top down.

Each module is laid out in an outline format.

A control structure encompasses the lines that are indented under it (control structures usually end with an ending keyword).

Variables and data structures are only implied.

Entries should describe actions performed without referring to the "how it's done" detail.

All major activities ought to be covered, and the modules should be worked out such that all relevant cases are covered.

If any single module becomes too cumbersome, it should be redesigned or broken into smaller modules.

(b) Level 2 Conventions

Each module created during Level 1 design should be included as a routine or function. All nonprimitive routines and functions used during Level 2 design should also be pseudo-coded.

Each section of pseudo-code (each routine) should include a list of variables and constants and a list of routines and functions that were used. This provides a cross reference.

Variables and data structures should be marked as "global" if they are used in more than one routine. The actual assignment of variables to specific routines may be done during Level 3.

Variables, constants, functions, and routines are written in upper case to make them distinguishable from control words.

Semicolons are used to separate "statements" from each other. There are no rigid rules however, for statement separation.

Use nested structures. They are much cleaner than branches. Indent the lines for readability.

Use miniscule letters to indicate specific elements in data structures. They may resemble arrays, but they make it easier to specify particular elements of lists. "ELEMENT-(i)" is an element of a list of elements, regardless of its actual format. "ELEMENT(I)" is an obvious array.

If you want to exit a loop prematurely, use generalized loop structures, such as:
"for each ITEM(i) do . . . next (i)"
"until CONDITION do . . . again"
"while CONDITION do . . . again"
"do (i) . . if CONDITION then exit (i) endif . . . again".

Instead of the "goto SOMEWHERE" structure, use the two conditional execution structures:
"if CONDITION then . . . else . . . endif"
"case (I) of (1) SUB1; . . . . else SUBn endcase"
(The "goto" is used only while coding, when the language lacks the sophisticated block structures or loop "exit" words.)

(c) Level 3 Conventions

The pseudo-code looks just like Level 2, except that data structures and variables are defined according to language and machine limitations.

Pointers can be initialized using the ADDR(X) function; "PTR1 :=ADDR(X)" is usually pretty clear.

Reference to a location via a pointer can be coded as [PTR1]. For example, "[PTR1] := X" copies the value of X to the location reference by PTR1.

Language-supplied constructs may be used; however, it is a good idea to limit these, as this is still part of the design phase.

Figure 4.

Table of Terms

primitive

A variety of subroutine or function that performs some very basic job within the context of your program. Disk access words, I/O routines, and data structure access words might be considered primitives. Of course, if you are writing a disk access package, you might use other routines that are even more primitive.

case statement

Something like an "indexed goto," where you execute one of several possible blocks of code depending on a selector variable.

loop structure

Any program control structure that allows a block of code to be executed repeatedly. This includes such things as "TOP: (some code) GOTO TOP;" that executes the code between "TOP" and the "GOTO TOP" statement.

global variable

Technically, a variable that may be directly accessed by name from all routines in a program. For our purposes, a variable that can be accessed by name from more than one routine. In block-structured languages, the degree of "globalness" of a variable depends upon the level of the routine within which it is defined.

data structure

Something like a collection of variables. Common data structures are: arrays, records, tables, linked lists, or files. The last type, however, usually implies disk-resident data.

module

A group of activities that can be taken together as a cohesive set. For our purposes, one of the functional subdivisions that helps describe the activity performed by the program. Modules interact with each other under the supervision of a main module.

top down design

A method of analyzing the design from the most general level, through increasing degrees of detail, until some bottom level of most intimate detail is reached. In our case, three levels of complexity occur before actual coding is reached, and most of the analysis occurs during the second level.

bottom up

The method of reconstructing the program from the lowest level of routines (the primitives) on up through the major modules. You analyze from the top down and build from the bottom up.

pseudo-code

A "fake" language designed to aid in the design process. It does what you want it to do, rather than forcing you to contort your thinking to fit its limitations.

tight (structured) English

A method of specifying an activity in a readable form. Rules for games or instructions for procedures to be followed are often written in structured English. It should get the idea across to a person, though it may not make much sense to most compilers. When you write using structured English, pretend that you are writing down a set of rules for someone to follow (as if they will be doing the activities of your program by hand).

More on Binary Magic Numbers

Edwin E. Freed presented some valuable information in his article "Binary Magic Numbers" (*DDJ* No. 78). Not only are the magic numbers useful, but the algorithms show how the parallel processing capabilities of even the simplest computer (in this case, the ability to process all of the bits of a "word" in parallel rather than one at a time) can be used to dramatically improve the execution time of a program.

He chose Pascal, however, to present his algorithms. I do not want get involved with the unsolvable debate about "Which is the best programming language?" but in this case I can confidently state that C is a better choice for the programs in the article.

The primary reason is that C has operators which map directly into machine operators, so there is no need to invent functions to mimic the hardware. The use of the operators &, |, ^, ~, << and >> which stand for AND, OR, EXCLUSIVE OR, COMPLEMENT, SHIFT LEFT, and SHIFT RIGHT, makes a C version of the program much more straightforward (at least to someone who can read C). The "op=" and "++" and "--" operators of C also correspond to the instructions available in assembly language, so I have used them.

In fact, in translating the programs to C in order to understand them more clearly, I was able to come up with cleaner versions of several of the functions presented by Freed. In most cases, I used a more natural index to control the loops. It also turned out to be useful to split the array B into two parts. B1 is the first half of the original array – the binary magic numbers. B2 is the second half – the complement of those numbers.

The resulting functions are shown in the listing (below). I hope this will direct the attention of people back to the ideas in Freed's article. You may never need to hand-code a parity function or a side-sum function, but someday you may need to sort or sum a matrix on a multiprocessor computer, and these or some other binary magic numbers may be just the trick you need in order to complete the task in log time rather than linear time.

■■J

by Dale Wilson

Dale Wilson, Codewright, 231 Couch Ave., St. Louis, MO 63122.

Binary Magic Listing

```
/* Some useful bit manipulation functions inspired by the article
 * "Binary Magic Numbers" by Edwin E. Freed in DDJ #78, April 1983.
 *   by Dale Wilson, 1983
 *   placed in the Public Domain
 *
 * These functions were written so they may be directly translated
 * into assembly language for most computers.
 *
 * These functions were tested on a Zenith 100 computer using the
 * C86 compiler from Computer Innovations, Inc.
 */
#include <stdio.h>


#define TRUE 1 /* stranger than fiction */
#define FALSE 0 /* fiction */
#define CNTLZ 26 /* MS-DOS eof */

#define N 16    /* bits per "word" */
#define V 4     /* log 2 of N */

/* Since C does not have binary constants, the binary magic numbers are
 * expressed below in hexadecimal.  B from Freed's article is divided
 * into b1 and b2 since there was no good reason to have them in the
 * same array.
 */
unsigned int b1[V] = { 0x5555, 0x3333, 0x0F0F, 0x00FF };
unsigned int b2[V] = { 0xAAAA, 0xCCCC, 0xF0F0, 0xFF00 };
```

```
/* converting binary numbers to Gray code is so simple that it may
 * be best defined as a macro rather than a function.
 */

#define binary_to_Gray(x) (x ^ x >> 1) /* X exclusive_or X shifted_right 1 */

/* MAIN routine to test the functions.
 *  Numbers (entered in hexadecimal) will be used as arguments in each
 *  of the functions.  As an additional check, the binary number resulting
 *  from the Gray_to_binary function will be converted back to Gray code--
 *  which should result in the original number.
 */

main(argc,argv)         int argc; char *argv[];
{       unsigned int r,i;
        int c;
        while(TRUE)                                     /* loop forever */
        {
                printf("Enter number : ");
                fscanf(stdin,"%x",&i);                  /* read a hexadecimal */
                while((c=getchar()) != '\n')            /* discard rest of line */
                        if(c == EOF || c == CNTLZ) /*  except on end of file */
                                exit();                 /*   quit */

                printf("low clear bit: %d\n", low_clear_bit(i));
                printf("high set bit : %d\n", hi_set_bit(i));
                printf("side sum     : %d\n", side_sum(i));
                printf("parity       : %d\n", parity(i));
                r = Gray_to_binary(i);
                printf("Gray code    : 0x%04x\n", r);
                printf("GTB To Binary: 0x%04x\n", binary_to_Gray(r));
                printf("Reversed bits: 0x%04x\n", reverse_bits(i));
                putchar('\n');                          /* leave a blank line between */
        }
}


/* This function returns the bit number of the lowest bit in the word
 * which is clear (zero).  The least significant bit is numbered 0, the
 * bit to the left of that, 1, and so on...
 * A minus 1 is returned for words in which all bits are on.
 * The time to execute this function is proportional to V which is
 * log2 of the number of bits in a word.
 * Note that the function low_set_bit may be created by complementing the
 * argument and calling low_clear_bit.
 */
low_clear_bit(value)    unsigned int value;
{       unsigned int temp;
        int i, result;
        if((value = ~value) == 0)                       /* complement, test for zero */
                result = -1;                            /*  zero means no clear bits */
        else
        {       result = 0;
                for(i = V-1; i >= 0; i--)
                {       result <<= 1;                   /* make space for next bit */
                        temp = value & b1[i];           /* test using magic numbers */
                        if(temp == 0)
                                result |= 1;            /* next bit of result is 1 */
                        else
                                value = temp;           /* discard disqualified bits */
                }
        }
```

(Continued on next page)

Binary Magic Listing (Listing continued, text begins on page 78)

```
        return(result);
}


/* This function returns the bit number of the highest bit in the word
 * which is set (one).  The least significant bit is numbered 0, the
 * bit to the left of that, 1, and so on...
 * A minus 1 is returned for words in which all bits are off.
 * The time to execute this function is proportional to V which is
 * log2 of the number of bits in a word.
 * Note that the function high_clear_bit may be created by complementing the
 * argument and calling high_set_bit.
 */
hi_set_bit(value)       unsigned int value;
{       unsigned int temp;
        int result, i;
        if(value == 0)                          /* if no bits are on */
                result = -1;                    /*   return that info */
        else
        {       result = 0;
                for(i = V-1; i >= 0; i--)
                {       result <<= 1;           /* space for next bit */
                        temp = value & b2[i];
                        if(temp != 0)
                        {       result |= 1;    /* next bit is one */
                                value = temp;   /* discarded unneeded bits */
                        }
                }
        }
        return(result);
}


/* This function returns a count of the number of bits which are on in a
 * word.  It executes in a time proportional to V, the log base 2 of the
 * number of bits in a word.
 * Note that a count of the number of zero bits in the word may be found
 * by complementing the value and calling side_sum.
 */
side_sum(value)         unsigned int value;
{       int i;
        unsigned int s;
        s = 1;
        for(i=0; i<V; i++)      /* use magic in reverse order */
        {
                value = (value & b1[i]) + ((value >> s) & b1[i]);
                s <<= 1;        /* generate the powers of two on the fly */
        }
        return(value);
}
/* This function converts a number expressed in Gray code to the
 * equivalent binary number.  It executes in time proportional to the
 * log base 2 of the number of bits in the word.
 */

Gray_to_binary(value)  unsigned int value;
{       unsigned int s;
        for(s = N >> 1; s != 0; s >>= 1)
        {
                value ^= value >> s;
```

```
        }
        return(value);
}
/* This function returns the parity of a word--that is, it returns a zero
 * if the number of one bits in the word is even, and a one if the number
 * of one bits in the word is odd.  The low order bit of the results of
 * Gray_to_binary and side_sum both have this property, so isolating this
 * bit gives the desired result.  Gray_to_binary was selected since it is
 * a faster function than side_sum.
 */
parity(value)           unsigned int value;
{
        return(Gray_to_binary(value) & 1);      /* build on previous work */
}
/* This function reverses the bits in a word.  Strangly enough, this turns
 * out to be a very useful function to have available.  Note that this function
 * works only on functions with word lengths which are powers of 2.
 */
reverse_bits(value)     unsigned int value;
{       unsigned int s,i;
        s = 1;                          /* s provides the powers of 2 */
        for(i=0; i<V; i++)
        {       value = ((value << s) & b2[i]) | ((value >> s) & b1[i]);
                s <<= 1;
        }
        return(value);
}
```

End Listing

A Sample Run:

```
Enter number : 0001
low clear bit: 1
high set bit : 0
side sum     : 1
parity       : 1
Gray code    : 0x0001
GTB To Binary: 0x0001
Reversed bits: 0x8000

Enter number : 0002
low clear bit: 0
high set bit : 1
side sum     : 1
parity       : 1
Gray code    : 0x0003
GTB To Binary: 0x0002
Reversed bits: 0x4000

Enter number : 0003
low clear bit: 2
high set bit : 1
side sum     : 2
parity       : 0
Gray code    : 0x0002
GTB To Binary: 0x0003
Reversed bits: 0xC000

Enter number : 0004
low clear bit: 0
high set bit : 2
side sum     : 1
parity       : 1
Gray code    : 0x0007
GTB To Binary: 0x0004
Reversed bits: 0x2000

Enter number : 8000
low clear bit: 0
high set bit : 15
side sum     : 1
parity       : 1
Gray code    : 0xFFFF
GTB To Binary: 0x8000
Reversed bits: 0x0001

Enter number : 7FFF
low clear bit: 15
high set bit : 14
side sum     : 15
parity       : 1
Gray code    : 0x5555
GTB To Binary: 0x7FFF
Reversed bits: 0xFFFE

Enter number : FFFF
low clear bit: -1
high set bit : 15
side sum     : 16
parity       : 0
Gray code    : 0xAAAA
GTB To Binary: 0xFFFF
Reversed bits: 0xFFFF

Enter number : 1111
low clear bit: 1

Enter number : 0000
low clear bit: 0
high set bit : -1
side sum     : 0
parity       : 0
Gray code    : 0x0000
GTB To Binary: 0x0000
Reversed bits: 0x0000

high set bit : 12
side sum     : 4
parity       : 0
Gray code    : 0x1E1E
GTB To Binary: 0x1111
Reversed bits: 0x8888

Enter number : 3333
low clear bit: 2
high set bit : 13
side sum     : 8
parity       : 0
Gray code    : 0x2222
GTB To Binary: 0x3333
Reversed bits: 0xCCCC

Enter number : 7777
low clear bit: 3
high set bit : 14
side sum     : 12
parity       : 0
Gray code    : 0x5A5A
GTB To Binary: 0x7777
Reversed bits: 0xEEEE

Enter number :
```

End Sample Run

SXR PLUS: Sorted Cross Reference

Company: Prasek Computer Systems, Inc., Box 2427, Santa Clara, CA 95055

Computer: Apple II/II+ (48K RAM) or IIe with DOS 3.3 and Applesoft One disk drive; *Optional (but strongly recommended):* Printer, 80-column card, second disk drive

Price: $39.95

Reviewed by Charles Petzold

SXR PLUS is a handy utility for programmers who need a cross-referenced index of variables in an Applesoft program. It is especially useful when debugging or modifying a program.

SXR PLUS provides 40- or 80-column output to a screen or printer. Output may be tailored to include or exclude line number references, numeric constants, and/or quoted literals (strings). A search feature is included to find specific variables. Special instructions are provided for programmers who want to move DOS into the upper 16K of an Apple IIe or to a 16K RAM card on an Apple II+.

Complete instructions are contained in a 24-page booklet. It's slow going the first time through because Prasek takes great pains to make sure you understand every step. After that first run-through, however, the procedure is so simple the manual isn't needed unless you start having problems. If that happens, a troubleshooting guide is included to put you back on track.

When SXR PLUS is booted, a menu-driven program is used to set initial parameters. You must decide whether you want line number references, numeric constants, and/or quoted strings included in the cross-referenced list. If you choose standard video output (a monitor or TV set), 40-column width is automatic. If you choose another output device (such as a printer), you must select the proper slot number and determine whether you want 40-column or 80-column output. The output will pause every 20 lines if you select the pause option.

Once the parameters are set, the SXR PLUS diskette is removed from drive 1 and replaced with the disk containing the program to be cross referenced. After this program is loaded, the disk is removed and the SXR PLUS disk reinserted. To get a cross reference, BRUN SXR PLUS and select either F for a full sort (as determined by the parameters) or S to search for a specific variable. That's about all there is to it.

SXR PLUS comes in handy when you're debugging. Good programmers keep a list of every variable they use. If you make a typo and accidentally use BA$ instead of AB$, BA$ will show up in a sorted cross reference, and you will have found the error and the line number where the mistake was made.

Another use for SXR PLUS is when you decide for one reason or another to change a variable. If you don't change every occurrence, you've got trouble. Using the search function, you'll be able to find (and later change) every occurrence.

Depending on the complexity of the program, a sort may take only seconds or it may take several minutes. For test purposes, I used the 47-sector LEMONADE program from Apple; a full sort took more than six minutes. I also tried a 91-sector program (PLANET.OF.THE.ROBOTS on a Softdisk Magazette), and a full sort took more than eight minutes. Both sorts were viewed on the monitor, not a printer.

Although SXR PLUS did everything it promised, there were some minor annoyances. To set the initial parameters, for example, you must answer six or eight questions, depending on your answers. After answering all questions, you are asked if the parms are correct. If they are not, you must repeat your answers to *all* questions. It would be quicker and easier to change only those answers that need correction (or updating). This can be easily fixed with some minor programming changes.

Another nice feature would be default parameters, with the default option being the option the user would most often select. To accept the default option, the user would merely hit the RETURN key.

Also, no provision is made for a two-drive system. With some more minor programming changes, this could be added. It would eliminate the disk-switching required under the one-drive system.

Since SXR PLUS is not on a protected disk, experienced programmers will no doubt be quick to make the necessary modifications.

With respect to copies and distribution, Prasek has what they call a "share-the-fare" program. You pay $39.95 for the first SXR PLUS package, and $7.50 for each additional package. There is no limit or restriction on the number of additional copies you can purchase at this price. User manuals alone are now $6.00.

If you destroy your disk, you can also get a replacement disk for $5.00 if you return the original disk. While the old warranty was only five days, you are now protected for 90 days. Returns are made to Prasek.

Perfect Writer Perfect Speller

Company: Perfect Software, Inc., 1001 Camelia St., Berkeley, CA 94710

Price: $399.00

Reviewed by Charles K. Ballinger

By now you probably have read innumerable reviews on the most common word processors currently on the market and still can't decide which one is for you.

Well, perhaps you are going about it the wrong way. As in most things you purchase, don't you decide what you want the product to do and then go out and find something that fits your requirements? (Of course, you know full well that in all things compromises must be made.)

It has been 5 months now since I first received this software for review. I have run it through its paces enough to feel comfortable with it, and I can now tell you what may make you interested in this product.

Since I do a lot of programming in a variety of languages, I look for a word processor that can handle the demands made on it by a special group of users (programmers) and still be used to produce the documentation that is so necessary for an item I would produce. The following are things I wanted in a word processor but had not been able to find in any of the others currently on the market:

1. No need for imbedded special codes that would prevent me from sending text files to other systems.
2. Ability to view more than one input file in order to merge more than one program source into a new program.
3. The absence of memory restrictions on the size of the source that I could edit at one time.
4. The ability to produce documentation that consists of several files and to print them in a contiguous fashion and also produce more than one copy at a time.

I found that Perfect Writer lives up

to these qualifications and even more.

As supplied (on 8-inch disk), the software was fairly easy to install. Perfect Writer is written in C and, as such, provides the authors a means to continually update the product. Prior to installation you should read the introduction to get an overview of the software. Pay close attention to the information on swapping (this is how Perfect Writer allows files larger than available memory), because this is where your disk capacity may play an important part. After the introduction you can proceed to the installation guide in Appendix A.

The terminal configuration portion is by far the easiest. If you have a KayPro, Apple, Vector, Televideo, IBM PC, Osborne, Heath-Zenith, Columbia, Access Matrix, Seattle Computer, GE Intersil, Superbrain, or NorthStar Advantage it is just a matter of selecting the terminal from a displayed menu. Since I am running a Heath H-8 system, it was a simple matter to select my terminal type.

If you can't find your terminal type on the menu then you may go through a question/answer session with the program to arrive at the correct terminal configuration. This is one of the first programs I've seen where they make it perfectly clear just what you are supposed to respond with. The questions that require other than a yes or no response tell you explicitly to enter your response in hex or in decimal as required, instead of leaving it up to you to guess whether you enter hex, decimal, octal, or ASCII as a response.

Perfect Software's attention to detail in the installation section was a real treat compared to some I've seen. Software I've installed usually falls down in the printer installation portion: not Perfect Writer, however. If you do not have one of the supported printers (Epson, IDS460, Centronics 737, Centronics PS, Diablo 1610, 1620, 1640, 1650, 630, PS or equivalent, NEC 5510, 5520 or PS), you can go through a question/answer session and define your printer to the program. This procedure is very easy to follow, even for a layperson or first-time user.

The documentation may, at times, seem overwhelming, particularly when you first start to use the software. Your best approach would be to skim the material and then start with the lesson disk to get an actual feel of the programs.

My only minor complaint with the documentation is in the size of the book. I hate having to break the spine of a book just to get the thing to lie flat so I don't keep losing my place (or having to place a weight on it to keep it open). A loose-leaf book, even if in the same size, would have been far easier to use and would have made updates or corrections to the manual possible. How do you correct a manual that is bound, short of making the user purchase an entirely new copy?

The associated fold-out reference card is very handy and goes a long ways in helping you find that command you are looking for. All in all their documentation rates a "well done."

With the inclusion of a separate lesson disk, it is almost impossible not to gain a working familiarity with Perfect Writer/Speller in a couple of hours. Granted you won't know all the commands or capabilities, but then of what value is a software product that you can take to its outer limits in such a short period of time? I'll tell you, it's probably one that you will replace very soon. Most of the commands have a scheme that is quickly apparent, and you'll find that they are easily memorized. What's nice about Perfect Writer/Speller is that I'm still discovering things that it can do.

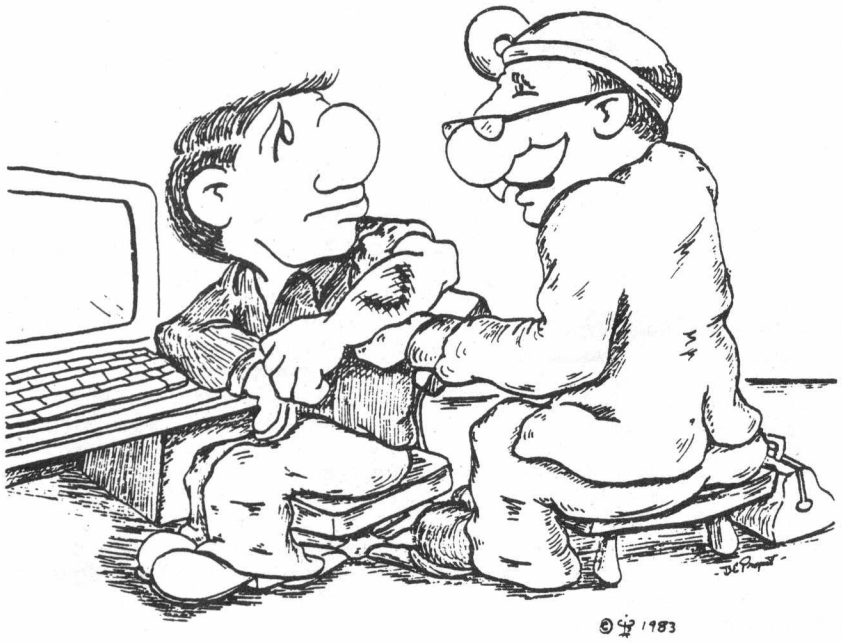
Perfect Speller contains a 50,000 word dictionary which is far easier to use than MicroPro's SpellStar. The speed, at least on my machine (2 MHz 8080 with 8-inch drives), approaches the estimate given in their manual: approximately 4000 words/minute.

I only had one minor problem with Perfect Writer/Speller (my own mistake, caused by being overzealous). When I contacted Perfect Software, Inc. – they even talk to users, unlike some – I found them to be most helpful and willing to provide the assistance I needed, pointing out that I had inadvertently missed a step. Not having identified myself as a software reviewer, I can only conclude that this is the normal support level that all users can expect to receive.

As mentioned earlier, this product is a true help to a programmer when it comes to constructing new programs from existing ones. The ability to have multiple file buffers is ideal from a programming point of view because you can now pull currently running source code into your new creation without undue effort. In fact, you are allowed access to a maximum of seven input documents, and, with the ability to use split-screen mode, new program construction time should be cut by a considerable margin.

All things considered I find this a perfect programmers' tool. It provides you with a product that will assist you from coding through final documentation of your software. Now you only have to learn one product to produce both your source code and your finished documentation, without the fuss and bother of trying to remember more than one set of word processing commands. I'm sure you have had the problem of mixing commands if you currently use more than one word processor or have access to different machines.

I heartily recommend this product as an excellent programming tool and documentation aid. With the introduction of the additional Perfect software family, and the trend of software vendors to provide integrated software, I think this is going to be a hard product to beat.



"Worst data byte I've seen in a long time,"

DDJ

BOOK REVIEWS

Cryptography. Proceedings of the Workshop on Cryptography, Burg Feuerstein, Germany, Mar 29-Apr 2, 1982.

Edited by Thomas Beth
Published by Springer-Verlag, 1983
$18.50, 402 pages
Reviewed by Richard L. Lozes

A Proceedings volume must be viewed from two perspectives: global and detailed. A smooth, logically connected, and well-styled global view is the responsibility of the editor. An accurate and complete paper is the responsibility of the authors.

These Proceedings cover a wide range of topics from classical cryptography to the Rivest-Shamir-Adleman scheme. Given the strides made in cryptography in the last decade, it is surprising to find papers concerning mechanical contrivances such as the Enigma machine. This historical view, nonetheless, is helpful; it motivates the abstract mathematical arguments later in the volume by means of physical analogies. The editor has performed his job ably.

The authors, too, hold up their end of the bargain. By and large, the papers are well organized and clearly presented. Understandably, one or two authors have difficulty with the English language.

Certainly, a Proceedings volume can never be recommended as a self-study text. However, I suspect that students of cryptography, with little prior exposure, could glean much valuable knowledge, especially if they were careful to study and commit to memory the introductory mathematics to be found in the first paper.

IBM Data Files

By David Miller
***Published* by Reston Publishing/Prentice-Hall**
$15.00 paper, 260 pages
Reviewed by James Moran*

One of the failures of documentation for personal computers is the paucity of adequate information on the use of disk files. Documentation for the IBM PC is not an exception, unfortunately, but puzzled PC users might find some needed answers in this book.

In *IBM Data Files*, author David Miller has created a useful tutorial for novice users of the IBM PC. The book assumes that the typical PC owner's familiarity with disk devices stops with the user's ability to load a piece of packaged software, and it attempts to address that limited knowledge in classic textbook format. Information is presented clearly in a precise manner and is followed by short quizzes to test the reader's understanding. One of the nice touches in the book is the grouping of programming examples at the end of each chapter. After finishing a chapter, the reader keys in the examples and runs them on the PC. In addition to reinforcing the learning process, this concept leaves the reader with a library of practical and usable programs after the book has been read.

IBM Data Files begins with a short introduction on the mechanics of using a diskette drive and progresses to file design, programming techniques, and, finally, file planning strategies for integrated systems – a particularly useful section, if a bit short. VisiCalc users will find the section on creating DIF files to be the most useful part of the book since the author was most generous in furnishing examples on how to standardize user file formats so that they may be transferred for use in a VisiCalc program.

This book does have some minor weak spots, and those would be most noticeable to readers towards whom author Miller seems to have directed his information: the beginning user. For example, it is somewhat surprising not to find a single diagram or graphic to visually enlighten the reader about the physical organization of a diskette. Some examples of random access algorithms would also have made this book more useful to PC owners who have an intermediate level of knowledge about their equipment.

All things considered, the book would probably be most useful to beginning and intermediate users. Those who are more technically knowledgeable may prefer a book with greater depth or more advanced topics although even they might find that the working program examples are worth the price of the book. Among those programs is a complete home inventory system that is contained in 36 pages of BASIC code. And for those readers that would rather not spend more than a few hours keying in the programs, an offer by the author to supply the programs on diskette for fifteen dollars seems like a pretty good deal.

*Copyright © 1983 by Compu-Syn.

Microprocessor Support Chips: Theory, Design, and Applications

By T. J. Byers
Published by MicroText/McGraw-Hill, August 1983
$38.00 hardcover, 224 pages (170 illustrations)
Reviewed by David W. Carroll

Microprocessor system designers have recognized that in most applications it is neither practical nor efficient to have a microprocessor perform all of the routine tasks required in a complex system. Rather, specialized support integrated circuits (ICs) have been developed to handle individual system support requirements.

Recognizing that these parts are necessary for economical, high performance designs in today's complex microprocessor-based systems, T. J. Byers has compiled a collection of 97 state-of-the-art support chips in his book, *Microprocessor Support Chips: Theory, Design, and Applications*, recently published by MicroText/McGraw-Hill. This design reference offers often hard-to-get information on many support ICs, including pin-outs, typical application schematics, and specific interfacing and design information. This book is not intended to replace the manufacturers' data sheets; rather it supplements them with real-world application information.

Some areas covered include telecommunications, power supply and special purpose, interface, control, video, and A/D and D/A converters. Telecommunications parts include various serial communications interfaces, protocol converters, local area network controllers and interfaces, and modem chips. Under power supply and special purpose we find keyboard and display controllers, data encryption chips, timers, parallel interfaces, and pulse width modulators for switching power supplies. The control ICs section includes printer and stepper motor controllers, floppy disk formatters and controllers, and Winchester hard disk controllers. The video chapter covers most currently popular

video display and CRT controller chips. Twelve data converters are included in the last chapter on A/D and D/A converters.

This book is primarily a design reference for serious digital engineers, allowing a rapid comparison of alternative components and helping to rapidly initiate the design of an operating application circuit. Each part is covered on two facing pages – details of operation, pin-out, and application notes are on the left page while the application schematics are on the right. The operating specifications and parameters of parts are not included in this book. Users should plan on obtaining data sheets on parts from their manufacturers. The book's format is hardcover, 8½ x 11 inches, making the pin-outs and schematics easy to read and somewhat justifying its $38.00 price tag, if one keeps in mind the limited audience and short useful life of this type of subject material.

The author advises that the selection of parts was made based on the current trends in the industry and the publisher's limitation of 100 total parts. A second volume may be expected next year, covering an additional 100 support ICs. grated circuits.

Overall, *Microprocessor Support Chips* lives up to its author's stated goal to "greatly facilitate the use of new chips in current designs." However, I find its subtitle, *Theory, Design, and Applications*, misleading. Very little theory is presented and design aid is mostly by example. Perhaps *Design Applications* would have been a more appropriate subtitle for *Microprocessor Support Chips*.

Z80 Applications

By James W. Coffron
Published by SYBEX
$15.95, 295 pages
Reviewed by Chuck Ballinger

As you might have guessed from the title, this book deals with the internal world of the Z80 chip as well as with all of the associated support chips.

The book is easy to follow and presents the concepts and functions of the Z80 in an easy to understand manner. With the aid of schematics and diagrams the author explains many areas of common confusion.

The book first takes you on a tour of the various ROMs (ROM, PROM, EPROM, and EAROM) and gives an in-depth explanation of each one in terms of function and application. Continuing on the tour, both static and dynamic RAM are explained, along with a discussion of circumstances where one would be preferred over the other. Complete layouts are also presented for most of the currently used RAM chips, including wiring hints.

Chapters on Z80 I/O and Interrupts proved very informative. I never really quite understood Interrupts until going through that chapter. The author's blend of excellent text with visual diagrams and circuit layouts presented what I had been unable to find in any previous text.

If you are interested in knowing the how and why of your Z80-based system, you are sure to find an answer in this book. In the chapters dealing with the Z80 PIO (8255), the author has even included assembler source for various routines that are common for the 8255 I/O chip; he steps you through the routines so you can see why they are coded in that fashion. This section also includes an explanation of the Z80 CTC (counter-timer-chip), as well as assembler source.

With the current interest in modem communications, the chapter on serial communications may help you explore the hows and whys. Start bits, parity bits, and stop bits are described clearly and thoroughly. For communications buffs the 8251 USART is covered pin by pin. Examples are presented that even provide information on how to send commands to the command register, which is something I've never seen explained to my satisfaction in any other book.

The book can be used for those interested in designing their own systems. However, I think that the number of homebrew systems being built are going the same direction as homebrew ham equipment – it's cheaper to buy an assembled unit than to build one. Where the book does come through is in the explanation of the various chips that make a Z80-based system and how special features found in most computer systems can be programmed.

The discussions of interrupts and programming the command modes for the serial I/O chips make the book a valuable addition to anyone's library. With the advent of newer CPU's many say the 8080 and Z80 will disappear. To the contrary, I say that you will start seeing the devices in your toasters, blenders, etc., and perhaps you would like to know what makes them tick. This book can be used both as a reference manual and a repair manual since it gives you an in-depth working of what goes on under your CPU's cover.

Concepts for Distributed Systems Design

By Gregor von Bochmann
Published by Springer-Verlag, Berlin-Heidelberg
$19.00, 259 pages
Reviewed by Robert Irving

Von Bochmann uses very formal guidelines when writing about systems. A full half of the book, including the annexes, is devoted to the requirements for specification of a distributed system. The first quarter defines and gives examples of distributed systems, specifically excluding multiprocessor systems but including a modular system in one physical location, which is laid out so that it could be distributed. The second quarter reviews architecture and communications protocols.

The reader is assumed to be a computer professional well-versed in conventional computer architecture and modular software design. This approach, combined with the terse, formal text and lack of an index, limits the application of this book as a text as well as a reference volume. However, if you already have a copy of Paul Green's *Computer Network Architectures and Protocols* (Plenum 1982), this book would be a useful supplement with regard to writing system specifications.

An Introduction to Numerical Methods with Pascal

By L. V. Atkinson and P. J. Harley
Published by Addison-Wesley Publishing Company
$16.95, 300 pages
Reviewed by Robert Ashworth

This book blends numerical methods with Pascal and calls upon the benefits of those data types to provide a natural implementation of the respective methods. While it is a college text for those of a scientific bent, no knowledge of numerical analysis is required. Familiarity with Pascal is assumed, and the text flows very well and can quickly be adapted for micros or even mainframe environments.

The text begins with a review of Pascal and builds on its merits: transparency (the intention of a well-written program being self-evident), security (the com helping to detect errors), and efficie (its design taking implementation into account). Emphasis is on the implementation of methods on a computer so as to reinforce their understanding.

Next comes a full treatment of rounding errors: those contributed by the method selected, and those inherent in the computer itself. Chapters 3-5 cover the solution of linear and nonlinear equations. Here the LU decomposition techniques are given prominence and the inverse of a matrix is explained. This part concludes with the treatment of the eigenvalue and the eigenvector problems, as well as a short discussion of the non-

symmetrical cases.

The next three chapters deal with discrete functions, differentiation and integration, and finally present an overview of ordinary differential equations. The Pascal programs are written out in full, extending over several pages.

Exercises were provided for each chapter but a solutions appendix was not included. I found the text to be very well written, with sixty-two useful programs.

Augmented Transition Networks

Edited by Leonard Bolc
Published by Springer-Verlag, 1983
$29.00, 213 pages
Reviewed by Chuck Ballinger

Confused by the title? Well, if you are not sure what an Augmented Transition Network is I'll try to explain. An Augmented Transition Network (ATN) is used in the design of interpreters, compilers, and editors as a method of factoring the input. Assume you have a data base that contains information on airplanes and you have presented a computer terminal with the following question: "How many Skyhawks required engine repair in 1973?" The processing of that request would require three main steps: parsing, interpretation, and evaluation. The first phase is what this book, as a collection of four papers, attempts to explain.

As a collection of papers that are related, but not directly integrated, there is some discord in the authors' presentations. Due to the nature of the subject, the amount of deviation does not detract from the initial subject, provided you are well versed in ATNs before picking up this book. This is not a tutorial or step-by-step book. If you don't know ATNs at the onset this book will not help you pick up any additional information.

Unless you are well versed in compiler design and implementation this book will be far above the average- or medium-level programmer. This text is intended for a very limited audience with a very specialized background or interest level. Since it is a collection of four papers, and each paper has little cross-relationship with the other, you must understand the material from the onset in order to have any inkling of what the individual authors are trying to convey.

If you are involved in the design of compilers, interpreters, or editors then this book may be of value to you. The textbook value may justify the cost as the areas covered include, "The Planes Interpreter and Compiler for ATN Grammars," "An ATN Programming Environment," "Compiling ATN into MacLisp," and finally "Towards the Elastic ATN Implementation." If one of these individual papers is of interest to you then you might be interested in this book. For the vast majority of computer users/programmers this book should be bypassed. For those interested, the book is full of examples of the various methods of ATN. The entire text is double-spaced in typical college manuscript format and is well structured.

■■J

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Bill Savage's floating-point benchmark program, discussed in this column in *DDJ*, September 1983, apparently caught the interest of a large number of readers. Versions in BASIC (9/83), PL/I (9/83), Fortran (1/84), Forth (11/83), Pascal (11/83), and C (Listing, page 96) have been contributed along with many timing reports.

In Table I (page 93) I have compiled the figures sent to me to date and sorted them in order of execution time (thank goodness for dBASE II). When duplicate timings on the same language and CPU were sent by different readers, I generally picked the more conservative results. A few languages were omitted altogether because of gross conflicts between two different reports (the IBM 4341 Fortran was one of these). I have also attempted to screen out any obvious typographical errors or blatant publicity grabs; nevertheless, let the reader beware!

The largest number of contributions came to *DDJ* via Gerald Edgar of Columbus, Ohio, who put the original benchmark program onto the CompuServe bulletin board system and asked for donations. This resulted in a set of over 50 timings, which he then sorted by degree of error, CPU, and language. After eliminating the duplications, I added these to my own table and designated them by the name "COMPUSERVE" in the "Source" column. Hearty thanks to all the anonymous CompuServe users who took the time to read and comment on this project.

Turning our attention to the table, the column "FPP" contains the type of hardware floating-point support that was used, if any was noted by the contributor. Most of the timings were accurate only to the nearest second (except for the very fast mainframes); they are displayed to three decimal places as an artifact of the dBASE II report program. The column "Error" was calculated as ABS (2500.000 - A), converting to scientific notation and rounding the mantissa to the nearest integer. Obviously, the smallest errors correspond to the most accurate results.

This collection demonstrates very clearly that personal computers based on the Intel 8086 or 8088, when augmented with the 8087 floating-point coprocessor, can offer performance on this limited benchmark that is competitive with "super-minis" or mainframe computers costing ten to a hundred times more. Of course, many significant issues such as the cost, size, and speed of mass storage, sophistication of systems tools, and ability to handle large numbers of interdependent processes are being totally ignored here.

We also received some advice and gentle chastisement from readers who are more knowledgeable on the subject of floating-point libraries and benchmarks than yours truly. I have excerpted a few of these letters below.

Jeffrey M. Speiser, of San Diego, California, writes: "It should be noted that for this benchmark the accuracy of the result is only suggestive of the accuracy of the floating-point arithmetic since:

"1) Errors of opposite signs can cancel in the final result, suggesting greater than actual accuracy.

"2) The errors can occur primarily over a small part of the argument range, suggesting much worse accuracy than is present over most of the argument range.

"3) Careful examination of one case showed that not only was the error primarily in the TAN-ARCTAN pair, but it could be explained by assuming that both functions were computed exactly, with roundings only to the nearest machine-representable number. Moral: do a simple sensitivity analysis for any critical calculation, by using the first two terms in a Taylor series expansion."

Harry J. Smith, of Saratoga, California, writes: "...I duplicated [the previously published results] for Fortran-80, BASIC-80, and PL/I-80 with the standard floating-point libraries. But when I also displayed the results of each iteration, I made a startling discovery. Both the Fortran-80 and BASIC-80 programs reached their final value of 2304.863 and 2304.860, respectively, on the 2230th iteration and then repeated this same output for the next 270 iterations. If the original program had been asked to do 2305 iterations instead of 2500, then Fortran-80 and BASIC-80 would have looked quite good. The foregoing exemplifies the possibility of superficial tests leading to false conclusions..." He went on to describe in detail some procedures for evaluating error in floating-point libraries, which we will save for a later column.

Karl J. Casper, Professor of Physics at Cleveland State University, contributed timings on machines ranging from the IBM 370 to the Hewlett-Packard 15C calculator. He commented: "This program... is somewhat unfair to the microcomputer if only single-precision numbers and functions are used. For example, I do not think that Microsoft really improved their algorithms from the Model I TRS-80 to the Color Computer. It is more probable that the increased accuracy results from increasing single precision to nine places on the CoCo.

"The problem is that the program begins to check angles very close to 90 degrees after only a few iterations. Most rational number approximations are designed to give good values for the functions over a wide range of arguments. It is certainly not clear that this program would correspond to any physically meaningful problems. Where physicists find that they are dealing with very small numbers, they would not usually try to evaluate these numbers by adding and subtracting large numbers. Physically meaningful benchmarks, however, would be very useful for evaluating computers.

"The point I am making is pretty simple. The Model I TRS-80 fails to get an accurate value simply because it lacks double-precision functions in BASIC. Both Pascal-80 and Fortran-80 (using double-precision functions) generate the correct value when run on the Model I, whereas Molinerx 5.1 Pascal generates the same value as Level II BASIC. (It is, of course, comforting to find that both the Sinclair ZX-81 and the HP-15C calculator obtain accurate values.) For a scientist, speed is occasionally important, but the fact that some computers run at breakneck speed is hardly significant when the wrong answer is obtained. Checking this benchmark on the IBM 370 yielded even faster speeds than MicroFloat [hardware-assisted floating-point libraries]. But while an interactive Waterloo BASIC obtained the correct answer, IBM BASIC failed by a wider margin than any microcomputer."

I will continue to add results to the data base as they come in and will plan to publish the expanded collection in about six months. If any readers have proposals for a more fair, exhaustive, and physically

meaningful floating-point benchmark program, let's see those too! We have a noticeable deficiency of C language results compared to the huge number of compilers that are currently available. Some interesting languages are missing altogether, including LISP, Logo, Ada, and Modula II. If you are kind enough to send in timing results, be sure to note whether single- or double-precision were used and the type of hardware floating-point processor (if any). ■DJ

Table 1.

Floating-Point Benchmarks for DDJ

COMPUTER	MHZ	LANGUAGE	VERS	FPP	TIME (SEC)	ERROR	SOURCE	COMMENTS
HONEYWELL		MULTICS FORTRAN			0.500	4E+1	SHERMAN GROMME	SINGLE PREC, TIME APPROX.
IBM 370		WATERLOO BASIC			0.570	2E-1	KARL CASPER	
DEC VAX 11/780		VMS FORTRAN-77			0.578	<1E-3	SHERMAN GROMME	SINGLE PREC
HP 1000-F		FORTRAN 4	2040		0.630	2E+2	COMPUSERVE	
IBM 3081		PL/I			0.660	2E-25	COMPUSERVE	
HONEYWELL		MULTICS FORTRAN			1.000	1E-3	SHERMAN GROMME	DOUBLE PREC, TIME APPROX
DEC VAX 11/780		VMS FORTRAN-77			1.054	<1E-3	SHERMAN GROMME	DOUBLE PRECISION
IBM 370		IBM BASIC			1.530	2E+3	KARL CASPER	PRESUMED SINGLE PREC
HP A700		FORTRAN 4			2.000	3E-8	COMPUSERVE	
IBM PC (8088)	4.77	WL SYSTEMS FORTH		8087	3.200	1E-3	JOHN GOTWALS	
8086	5.0	PL/I-86	1.01	8087	3.700	2E+1	BILL SAVAGE	MICROFLOAT LIBRARY
HP 9000 (68000)		FORTRAN (UNIX)			4.000	2E-6	COMPUSERVE	
HP 1000-F		FORTRAN FTN4	2040		4.000	5E-3	COMPUSERVE	
8086	5.0	PL/I-86	1.01	8087	4.000	2E+1	COMPUSERVE	
IBM PC (8088)	4.77	MVP-FORTH		8087	4.300	<1E-3	C. SPRINGER	
DEC PDP 11/44		RSX-11M FORTRAN		FIS	4.500	2E+2	JOHN TOSCANO	
DEC VAX 11/780		PASCAL	2.1		5.000	6E-10	COMPUSERVE	
IBM PC (8088)	4.77	MICROSOFT PASCAL	3.11	8087	6.000	<1E-3	J. SPEISER	DBL PREC, SEATTLE 87.LIB
IBM PC (8088)	4.77	MICROSOFT FORTRAN	3.1	8087	6.200	<1E-3	J. SPEISER	DBL PREC, SEATTLE 87.LIB
DEC PDP 11/44		RSX-11M FORTRAN		FIS	6.900	2E-1	JOHN TOSCANO	DOUBLE PREC
DEC PDP 11/34		RT-11 FORTRAN		FIS	7.500		J. SPEISER	
IBM PC (8088)	4.77	BASIC COMPILER	1.00	8087	7.800	1E-3	J. SPEISER	DBL PREC, SEATTLE 87.LIB
HP 1000F		BASIC			8.000	3E+2	COMPUSERVE	
8085	5.0	RMAC		8232	10.200	5E-3	BILL SAVAGE	MICROFLOAT LIBRARY
8085	5.0	PL/I-80	1.40	8232	10.400	5E-3	BILL SAVAGE	MICROFLOAT LIBRARY
8085	5.0	BASIC-80	5.20	8232	10.700	<1E-3	BILL SAVAGE	MICROFLOAT LIBRARY
8085	5.0	FORTRAN-80	3.40	8232	12.500	5E-3	BILL SAVAGE	MICROFLOAT LIBRARY
IBM PC (8088)	4.77	BASIC COMPILER	1.00	8087	14.200	<1E-3	J. SPEISER	DBL PREC, MICROWARE LIB
6502	4.0	UCSD P SYSTEM	II	8231	15.500	2E+2	STEVEN SPEARS	
8088	4.7	UCSD PASCAL	IV.1	8087	18.000	2E-7	COMPUSERVE	
APPLE II (6502)	1.75	MVP-FORTH		9511	18.500	2E+2	C. SPRINGER	
DEC PDP 11/44		RSX-11M BASIC-PLUS-2			21.800	1E+2	JOHN TOSCANO	SINGLE PREC
DEC MINC 11/23		FORTRAN		FPU	22.000	2E+2	BILL SAVAGE	
DEC PDP 11/44		RSX-11M BASIC			27.200	1E+2	JOHN TOSCANO	
HEATH H-89 (Z-80)	4.0	FORTRAN-80		9511	31.000	2E+2	JOHN TOSCANO	
DEC PDP 11/44		RSX-11M FORTRAN			34.600	1E+2	JOHN TOSCANO	
DEC MINC 11/23		MINC BASIC	2.0	FPU	38.000	2E+2	BILL SAVAGE	
HP 9186		HP BASIC			44.000	3E-7	COMPUSERVE	
HP 9836 (68000)		BASIC			44.290	<1E-3	BILL SAVAGE	
IBM PC (8088)	4.77	BASIC COMPILER	1.00		48.000	9E+1	J. SPEISER	SINGLE PREC
DEC LSI-11/23		C (UNIX)	V7		56.000	6E-8	COMPUSERVE	
DEC LSI-11/23		PASCAL-2	2.1A	FP	66.000	1E-7	COMPUSERVE	
WANG PC (8086)	8.0	BASIC INTERP	1.0.2		68.000	2E+2	RAY DUNCAN	

(Table I continued)

COMPUTER	MHZ	LANGUAGE	VERS	FPP	TIME (SEC)	ERROR	SOURCE	COMMENTS
IBM PC (8088)	4.77	SUPERSOFT FORTRAN	1.07		70.000	9E+1	HUGH KAWABATA	SINGLE PREC
SUN WORKSTN (68000)		UNIX CC COMPILER			85.000	<1E-3	JOHN TOSCANO	
IBM PC (8088)	4.77	CI C86	1.33D	8087	85.000	1E-3	J. SPEISER	DOUBLE PREC
8086	5.0	BASIC-86	5.20		92.200	3E+2	BILL SAVAGE	
DEC PDP 11/44		RSX11M FORTRAN			103.000	2E-1	JOHN TOSCANO	DOUBLE PREC
DEC PDP 11/44		RSX-11M BASIC-PLUS-2			113.600	<1E-3	JOHN TOSCANO	DOUBLE PREC
68000	8.0	UCSD PASCAL	IV.12		115.000	3E-7	COMPUSERVE	
IBM PC (8088)	4.77	IBM APL			117.000	1E-8	J. SPEISER	
6809	2.0	PASCAL			119.000	8E+0	ANDY BALL	MICROWARE SYSTEMS CORP
HP 1000E		FORTRAN 4	2026		121.000	3E-8	COMPUSERVE	
8085	5.0	FORTRAN-80	3.40		140.800	2E+2	BILL SAVAGE	
8085	5.0	BASIC-80 COMPILER	5.20		140.800	2E+2	BILL SAVAGE	
HP 9835B		BASIC			140.800	1E-3	J. SPEISER	
ZENITH Z-100 (8088)		ZBASIC			142.730	2E+2	BILL SAVAGE	
6809	2.0	BASIC09			149.000	2E-2	ANDY BALL	MICROWARE SYSTEMS CORP
8088	8.0	SSS FORTRAN	1.04		154.000	2E-7	COMPUSERVE	
IBM PC (8088)	4.77	BASIC INTERPRETER			157.000	3E+2	J. SPEISER	SINGLE PREC
IBM PC (8088)	4.77	BASIC INTERPRETER	1.00		160.000	3E+2	J. SPEISER	DOUBLE PREC EXC TRANS.
IBM PC (8088)	4.77	BASIC COMPILER	1.00		170.000	1E-3	J. SPEISER	DOUBLE PREC
8086	8.0	MS PASCAL	?		171.000	1E-7	COMPUSERVE	
8085	5.0	BASIC-80	5.20		174.900	2E+2	BILL SAVAGE	
8086	5.0	PL/I-86	1.01		179.600	8E+2	BILL SAVAGE	
Z-80	4.0	BASIC-80	5.30		184.000	2E+2	COMPUSERVE	
HEATH H-89 (Z-80)	4.0	MBASIC COMPILER	5.2		189.700	2E+2	JOHN TOSCANO	
HEATH H-89 (Z-80)	4.0	FORTRAN-80			203.000	2E+2	JOHN TOSCANO	
TANDY 16B (68000)		XENIX CC COMPILER			211.000	1E-3	JOHN TOSCANO	
IBM PC (8088)	4.77	SUPERSOFT FORTRAN	1.07		211.000	<1E-3	HUGH KAWABATA	DOUBLE PREC
68000	6.0	C (TRS-XENIX)			212.000	1E-6	COMPUSERVE	
IBM PC (8088)	4.77	DRI PERSONAL BASIC	1.0		217.000	1E+3	RAY DUNCAN	SINGLE PREC
HEATH H-89 (Z-80)	4.0	MBASIC	4.8		229.800	2E+2	JOHN TOSCANO	
ZENITH Z-100 (Z-80)		MBASIC	5.22		236.740	2E+2	BILL SAVAGE	
8085	6.0	C/80	3.0		238.000	3E+2	COMPUSERVE	
DEC PDP 8		OS/8 FORTRAN IV			240.000	1E+3	SHERMAN GROMME	SINGLE PREC, TIME APPROX
8085	5.0	PL/I-80	1.30		254.400	8E+2	BILL SAVAGE	
IBM PC (8088)	4.77	DRI PERSONAL BASIC	1.0		255.000	1E+3	RAY DUNCAN	DOUBLE PREC
68000	6.0	BASIC/S-16	1.7		266.000	2E+2	COMPUSERVE	
HP-85 CALCULATOR		BASIC			280.767	1E-3	BILL SAVAGE	
HP-85 CALCULATOR		BASIC			281.000	6E-4	COMPUSERVE	
8088	8.0	COMP. INNOV. C86	1.33		288.000	7E-6	COMPUSERVE	
HP-75 CALCULATOR		BASIC			325.000	6E-4	COMPUSERVE	
Z-80	4.0	BASIC-E	2.2		330.000	5E+1	COMPUSERVE	
HEATH H-89 (Z-80)	4.0	C/80	3.0		370.700	3E+2	JOHN TOSCANO	
Z-80	4.0	PL/I-80	1.3		373.000	9E+2	COMPUSERVE	
OSBORNE I (Z-80)		C/80			420.000	2E+2	BOB BRIGGS	
TANDY I (Z-80)		FORTRAN-80			422.000	2E+2	KARL CASPER	SINGLE PREC
APPLE II (6502)	1.75	APPLESOFT			463.000	<1E-3	C. SPRINGER	
APPLE II (8088)	1.75	APPLESOFT BASIC			470.000	<1E-3	R. S. SCHLAIFER	
Z-80	4.0	NEVADA FORTRAN	2.2		473.000	8E+1	COMPUSERVE	
VIC-20 (6502)	1.0	BASIC	2.0		477.000	9E-5	COMPUSERVE	
TANDY I (Z-80)		MOLINERX PASCAL	5.1		485.000	1E+2	KARL CASPER	

(Continued on next page)

(Table I continued)

COMPUTER	MHZ	LANGUAGE	VERS	FPP	TIME (SEC)	ERROR	SOURCE	COMMENTS
TANDY I (Z-80)		BASIC			512.000	1E+2	KARL CASPER	
COMMODORE 64 (6510)		BASIC			514.000	<1E-3	TERRY THOMAS	
Z-80	4.0	ZBAS	1.4		540.000	2E+2	COMPUSERVE	
TANDY CC (6809)		FORTH			560.000	<1E-3	GARY BERGSTROM	
TANDY CC (6809)		BASIC RS-CC	1.0		585.000	<1E-3	GARY BERGSTROM	
ZENITH ZW110 8088	5.0	CI C86			655.000	<1E-3	A. F. HERBST	
IBM PC (8088)	4.77	CI C86	1.33D		695.000	1E-3	J. SPEISER	DOUBLE PREC
TANDY 16B (68000)		XENIX MBASIC			773.600	1E-3	JOHN TOSCANO	
IBM PC (8088)	4.77	BASIC INTERPRETER	2.0		890.000	1E-3	J. SPEISER	DOUBLE PREC
Z-80	2.0	NEVADA FORTRAN	3.0		975.000	8E+1	SHERMAN GROMME	
8085	6.0	CB-80	1.2		997.000	2E+1	COMPUSERVE	
8085	6.0	FORTRAN-80	3.4		1251.000	1E-12	COMPUSERVE	
8085	6.0	CBASIC	2.06		1623.000	2E+1	COMPUSERVE	
Z-80	4.0	BASIC-80	5.3		1980.000	1E-7	COMPUSERVE	
Z-80	4.0	AZTEC C II	1.05		2190.000	1E-5	COMPUSERVE	
HEATH H-89 (Z-80)	4.0	AZTEC C	1.05		2226.000	<1E-3	JOHN TOSCANO	
TI-59 CALCULATOR					3240.000	7E-3	COMPUSERVE	
Z-80	2.5	S-BASIC			3900.000	3E+4	COMPUSERVE	
TANDY I (Z-80)		FORTRAN-80			4602.000	<1E-3	KARL CASPER	DOUBLE PREC
TANDY I (Z-80)		PASCAL-80			5220.000	<1E-3	KARL CASPER	
Z-80	4.0	JRT PASCAL	2.1		5760.000	1E-3	COMPUSERVE	
HP-65 CALCULATOR					6000.000	9E-1	COMPUSERVE	
HP 15C CALCULATOR					10200.000	1E-2	KARL CASPER	
SINCLAIR ZX-81		BASIC			86400.000	3E-1	KARL CASPER	ABOUT 1 DAY

End Table I

16-Bit Toolbox Listing (Text begins on page 92)

```
main()
/* Floating point accuracy test adapted from DDJ 9/83 p. 122 */
{
  int i,iloop=2500;
  double a=1;
  double tan(),atan(),exp(),log(),sqrt();

  for (i=1;i<iloop;i++)
    {
     a=tan(atan(exp(log(sqrt(a*a)))))+1.0;
    }
  printf("\na = %10.4f\n",a);
}
```

End Listing

OF INTEREST

by Michael Wiesenberg

Faster than a Speeding 8088

Lightning 286, from Lomas Data Products, is an S100 80286 board that runs 8088 and 8086 software four times faster than a 5 MHz 8086 (which itself is considerably faster than the 8088), and can run the 80287 high-speed numeric processor, giving program execution *in some programs* a doubling of speed. The Lightning 286 can run at clock rates up to 10 MHz, and Lomas suggests that they will increase their rates as Intel improves the 80286 (which currently runs at only 6 MHz). Lightning 286 allows 16 Mb physical address space and 1 Gb virtual address space. Four operating systems, MS-DOS, CP/M-86, Concurrent CP/M-86, and MP/M-86, are currently available with the board, which conforms to IEEE 696 S100 bus specifications. The board costs $1395, with optional 80287 for another $550.

High Speed Modems

Prentice Corporation's limited distance, **ALD/1**, full and half duplex, asynchronous 1200 to 9600 baud modem conforms to Bell specifications 43401 and 41028 with metallic, twisted-pair Local Area Data Service (LADS) for local office and campus networks. It comes in a stand-alone version for $300 and rack-mountable for $200. The **HSLD/1**, 2400 to 19,200 baud, is $490 and $390. The synchronous leased line 4800 baud **208A/B** is $1750 and $1650, while the 9600 baud **9629** for four-wire 3002-type unconditioned leased lines in point-to-point applications is $2750 and $2650.

Here Come the Little Guys

Even though I rarely mention products for which the press release lists no price, I'd like to describe this one because it shows what you can shortly expect many computers to look like. The **UDI-500**, from Universal Data Incorporated, is the first portable computer with internal batteries and *two* disk drives. At 11 by 13 by 3 1/8 inches and 12.8 pounds, it is less than half the size and weight of competitors. The CMOS Z80 permits up to 12 hours of operation on one charge of the nickel-cadmium batteries. You can get 80 hours if you never use the disks or two with constant disk use. The batteries charge fully in 10 hours. The LCD has eight lines of 40 characters. An accessory slot accommodates the optional 300 *or* 1200 baud direct-connect modem. The low-power static CMOS memory is 64K standard, expandable to 256K. Two plug-in CPU boards (both CMOS) come with the computer, Z80 and 1805. You get CP/M 2.2 and Micro-DOS. Available software includes text processor, spelling checker, data filing system, spread sheet, communications package, and various flavors of BASIC.

Computers Talking to Each Other

SOFTCOM Telecommunications Utility, from the Software Store, is an intelligent terminal program and CP/M to CP/M file transfer utility that supports XON/XOFF and transmits at up to 9600 baud in full or half duplex. It runs on 8080, 8085, and Z80 systems with at least 32k and CP/M, and costs $150.

CP/M Plus for Cromemco

Super BIOS Plus, from MICAH (Micro Applications and Hardware), is CP/M Plus for Cromemco computers, starting at $495. You get source code for all modules. You can also add the **Expand** program, a CDOS emulator, and the Select Word Processor. MICAH also carries Morrow hardware configured for Cromemco. **Reader Service No. 115.**

Don't Give Me Any Static

Staticide Wipes, from ACL Incorporated, are individually packaged towelettes for cleaning and static control of video screens, packed in square foil packs, 24 to a box, $4.98 per box. ACL offers free samples and information on other Staticide products.

Come Forth

The Institute for Applied Forth Research is hosting the **1984 Rochester Forth Applications Conference** June 5 to 9 at the University of Rochester. Speakers will discuss real-time systems. The conference sponsors are calling for papers on Forth applications and techniques. A 200-word abstract is due April 1 and the paper should be in by May 1. Papers should be on real-time software (including process control, data acquisition, smart instrumentation, laboratory systems, robotics, computer vision, spacecraft navigation, music and voice synthesis), applications (in particular, Forth microchip applications), or Forth technology (finite state machines, control and data structures, and hybrid hardware and software systems). Papers should be sent to Diane Ranocchia (who can be contacted for more information), Institute for Applied Forth Research, Inc., 70 Elmwood Avenue, Rochester, New York 14611; (716) 235-0168.

Proportional Spacing on Wordstar

Proportional Spacing on WordStar, from the Writing Consultants, is a $20 book that tells you how to do what (they claim) "the industry has said WordStar needs," plus print two or more justified columns on a page and underline_spaces_between_words.

UNIX for the People

Will UNIX be on microcomputers? **Real UNIX? The Office UNIX System** from Unisource is not a lookalike, but full UNIX, licensed from AT&T, and developed as VENIX/86 by VenturCOM for the IBM PC. Unisource's Of-

fice Menu Tool uses prepared menus to run applications or execute commands, and you can make your own menus. You get four editors, C compiler, BASIC, assembler, uucp, cu, nroff, a spelling checker, table formatter, library routines, vi, more, termcap, and the c shell, in single user format for $800 and multiuser for $1000. The Office Menu Tool is $125. You can add the FinalWord for $395, Leverage DBMS for $385, ViewComp spreadsheet for $395. There are also various single and multiuser combinations, with VENIX/86 plus one or more applications, starting at $1125.

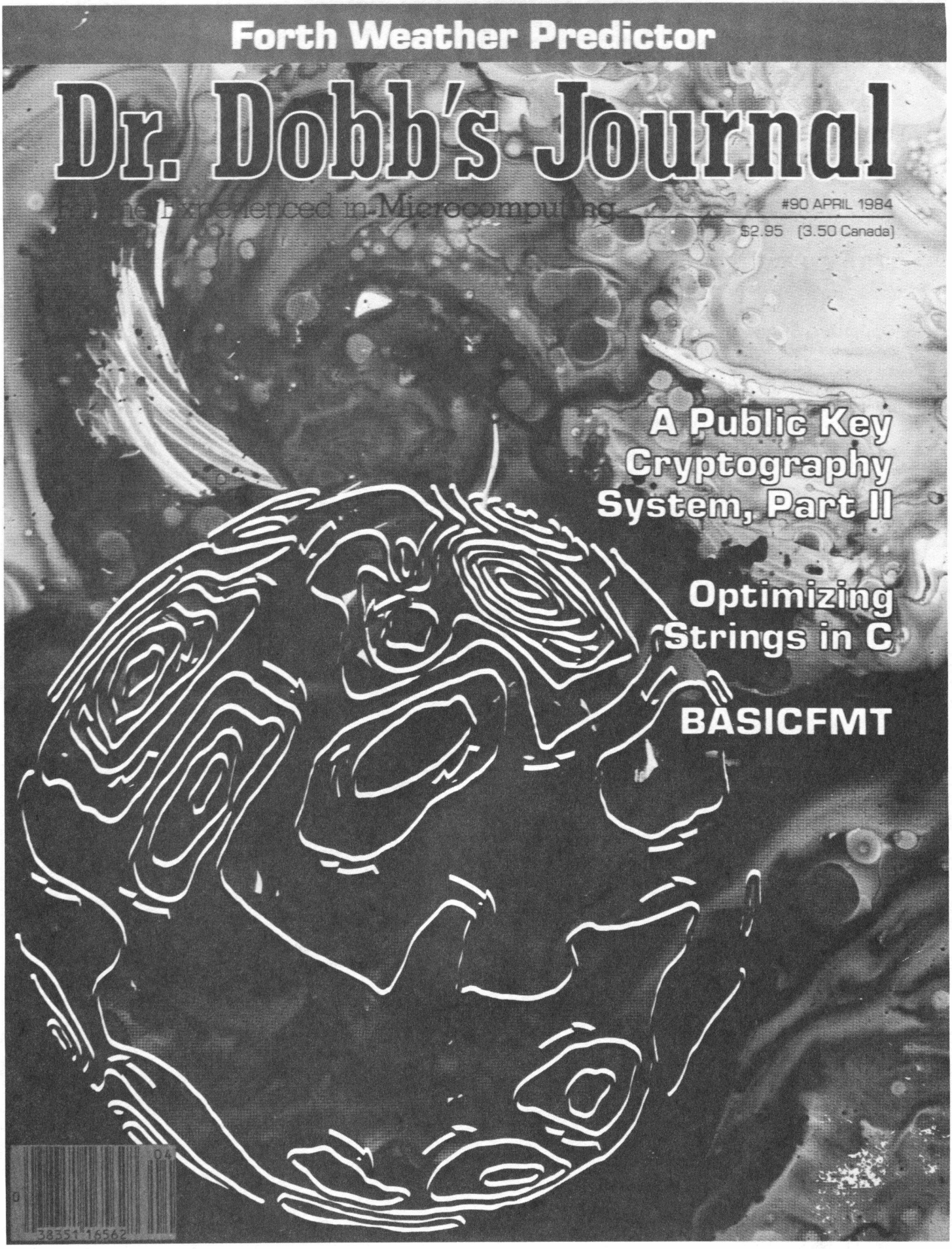
Fortran 77

Those of you who use Fortran 66 or, heaven forbid, Fortran 2 and are tired of the gibes from your friends who use Pascal ("Nyah, nyah, you don't have good string-handling capabilities." "Your language ain't structured.") will be thrilled to learn that Absoft has **FORTRAN 77** for the Whitesmiths Idris operating system on 68000 systems. This is an ANSI Fortran 77 compiler that generates position-independent and reentrant object code, supports virtual arrays, has a full screen source level symbolic debugger, and enables dynamic linking of modules at run time. It compiles up to 3500 lines of code per minute and has no restrictions on code size. You'll pay $695 to $2000.

And a Small Powerhouse

How about a portable computer with a 132-column screen? The **Execuport XL** from Computer Transceiver Systems Inc. is just that, with Z80 and CP/M, while the **Execuport XL+** has both the Z80 and 80186 (true 16-bit) and MS-DOS. Both come bundled with Perfect Calc, Speller, Writer, and Filer. Optionally, you can get OASIS, CP/NOS, CP/NET, and MP/M. The XL has 80K, while the XL+ has 128K, which can be expanded. Both have a 9-by-5 green phosphor screen with 132 columns by 24 lines, two double-sized, double density 5¼-inch drives with 800K storage each, and 22 user-programmable function keys. The XL is $2695 and the XL+ $3495. You can lease them for, respectively, $98 and $130 per month. The computers are supported by CTSI's nationwide on-site sales and service programs.

DDJ


Forth Weather Predictor
Dr. Dobb's Journal
for the Experienced in Microcomputing
#90 APRIL 1984
$2.95 (3.50 Canada)
A Public Key Cryptography System, Part II
Optimizing Strings in C
BASICFMT
38351 16562

Dr. Dobb's Journal

For the Experienced in Microcomputing

April 1984
Volume 9, Issue 4

CONTENTS

ARTICLES

18 Optimizing Strings in C
by Edward McDermott
The decision to provide new capabilities to one's compiler should be made carefully. Finding that C had no string-manipulation verbs, the author decided to optimize his compiler's string handling. In addition to his routines, this article examines some of the considerations involved in such a decision.

24 Expert Systems and the Weather
by Jack Park
The author chose predicting the weather as the subject of an early effort at knowledge engineering using EXPERT-2, a tiny, Forth-based, fifth-generation type language. With it, users can write high-level programs for performing logic inferences using rules written into their programs. The weather predictor presented here illustrates how a set of rules can be implemented or customized.

32 RSA: A Public Key Cryptography System, Part II
by C. E. Burton
In Part I of this article, the mathematical foundations of a Rivest-Shamir-Adleman public key encryption system were explored. This final installment discusses the advantages of a PKS, as well as the generation of the keys, encryption and decryption of text, and the use of "digital signatures" to assure authenticity of a message's origin.

60 BASICFMT for TRS-80 (A BASIC Formatter)
by Davy Crockett
No need to save a separate ASCII version of your BASIC program. This routine reads a BASIC program in compressed format from disk and prints a readable listing.

DEPARTMENTS

6 Editorial

8 Letters

10 Dr. Dobb's Clinic
Super Directory Fix; RAM-Drive Needed; The Great American Merchandiser; What's So Hard About CP/M Plus?; CP/M Plus, Apple and ALS; How It's Done; Our April Item

12 CP/M Exchange
CP/M V2.2 Compatibility; BDOS Function 10: Vastly Improved

76 Software Reviews
microSUB:MATH;METAFILE Application Development System

86 16-Bit Software Toolbox
More on MS-DOS EXEC Function; PC-DOS Manual Typo; IBM PC Junior; MS-DOS Volume Labels; Finding Size of TPA under MS-DOS 2.0; CP/M Discrepancies

94 C/Unix Programmer's Notebook
Comments about Unix; Low Level Input-Output in C

98 Of Interest

Publisher - Jane Nissen Laidley

Editorial
Editor – Reynold Wiggins
Managing Editor – Randy Sutherland
Contributing Editors –
Robert Blum, Dave Cortesi,
Ray Duncan, Anthony Skjellum,
Michael Wiesenberg

Marketing
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Marketing Coordinator – Ed Gueble

Advertising
Advertising Director – Beatrice Blatteis
Advertising Sales –
Alice Hinton, Walter Andrzejewski

Circulation
Circulation Director – Terri Pond
Circulation Assistant – Frank Trujillo

Production
Art Director/Production Manager –
Shelley Rae Doeden
Production Coordinator – Alida Hinton
Typesetter – Jean Aring

Cover Illustration – Inge Infante

Copyright © 1984 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, *drawn on a U.S. Bank.*

Donating Subscribers Contributing Subscriber: $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, recieves *five* one-year subscription).

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, Info World, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R. B. Sutton, Stan Veit. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines and the current compensation schedule by sending a self-addressed, stamped envelope.

Foreign Distributors UK & Europe: Homecomputer Vertriebs HMBH 282, Flugelstr. 47,4000 Dusseldorf 1, West Germany; Euro Computer Shop, 182 Rue du Faubourg St. Denis, 75010, Paris, France; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – premiere, F-75014, Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** AXCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian Distributors)*

EDITORIAL

Dr. Dobb's Journal began as a forum for sharing information and ideas about programming and computers. It continues to be a place to present new languages, utilities, tools, applications, algorithms, discoveries, and techniques to the microcomputing community. That community may then use, comment upon, improve, and expand these items. Many of the things explored within our pages have had major influence in the industry. *DDJ* readers have always been at the forefront of the microcomputer industry, providing a wealth of expertise to fuel discussion. Our authors have come primarily from within the readership, and it is this reader involvement that has sustained and guided the *Journal* through the years.

The material currently discussed in our pages is far more complex than it was eight years ago, and this has forced some adjustments. We continue to print complete listings of useful and innovative software, and to explore developments in areas such as languages and operating systems. Maintaining the community discussion of these larger, more complex ideas, however, has become increasingly cumbersome. We have periodically instituted reader-driven columns to provide more intimate exchange in areas of particular reader interest, the latest addition dealing with C and Unix. Most recently, we formed a board of referees (again, from within the readership) to help ensure that the articles published are as appropriate and accurate as possible.

From our homespun beginnings with tiny BASIC and tiny computers, we have grown with the microcomputing industry and community. Cosmetic changes have occurred to improve appearance and marketability. Fiscal considerations have prompted us to accept advertising, though we still work to keep the percentage down and to maintain our objectivity. And it was advertising revenue that helped us to begin paying authors within the last few months. Each move has been made because we felt it would allow us to better perform our function without compromising our traditions.

Now we have had a new opportunity presented to us – one which can open the door for many of the projects that we have longed to pursue, and one which will provide added resources with which to continue improving the quality of material we bring to you. This new opportunity will allow the tradition of *DDJ* to continue and allow exploration of new vistas.

We are joining forces with M&T Publishing, Inc., a company dedicated to quality and integrity, a company which likes what *DDJ* has been doing and would like to see it done even better. Laird Foshay, the president of M&T Publishing, was part of the *DDJ* staff in the past, and has always been interested in our objectives. People's Computer Company will still own *DDJ* while M&T will be licensed to publish the magazine. PCC's Executive Director will chair an advisory board set up to ensure that *DDJ* maintains its focus and direction. The current staff, slightly expanded, will continue to produce the magazine as usual.

Among the new faces will be Michael Swaine, who is joining the staff as editor-in-chief. Many of you may recognize the name from his three-year tour at *InfoWorld* as a senior editor. A long-time reader of *DDJ*, Mike is dedicated to the traditions of the *Journal.* A professional journalist and computer enthusiast, Mike is interested in continuing to improve the quality and scope of the service we provide. We are excited to have him on the team.

So what does this all mean? Well, it means that we will have a new address. It means new projects (perhaps a bulletin board here at the *DDJ* office) and new opportunities. It means better service to our subscribers and better communication links to our readers. It means more quality, more information, more of what you buy *DDJ* for. We will continue to explore ways to facilitate the dialogue among the microcomputing community. Don't look for radical changes in the *Journal;* we have no intention of diverting from our long-chosen path. Expect instead a continuation of our gentle evolution.

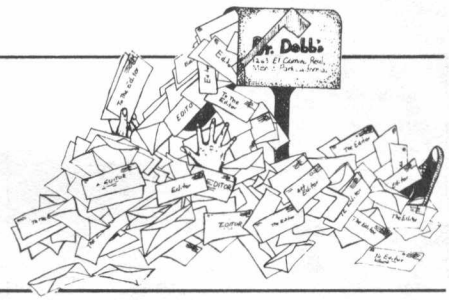
Reynold Wiggins

THIS MONTH'S REFEREES

A. Gomez, Telecomp, Inc.
Kim Harris, Forthright Enterprises
Richard Relph
John K. Taber, IBM
Bob Wheeler, Custom Hardware

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94026. Second class postage paid at Menlo Park, California 94026 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94026 · 415/323-3111 **ISSN 0278-6508**

LETTERS



NBASIC, Postprocessed

Mr. Shammas' NBASIC program should not be confused with the NBASIC program marketed by the Computer Toolbox, Inc. – Ed.

Dear Sirs:

In reference to the article about NBASIC on page 24 of your 1/84 issue, I suspect the program has the following errors:

(1) Line 1220. If a variable has 'REM' in it and it appears on a line before a Remark, it will be changed to a single quote.

(2) Line 1120. If a filename is entered without a period in it, then line 1120 never gets executed and the output filename FOUT $ never gets defined.

I have several preprocessors and use them a lot. I think they should be promoted more in all computer journals.

Sincerely yours,
Peter Ansbacher
918 South Rustic Road
Columbia, MO 65201

Dear *DDJ*:

I read the article "NBASIC: A Structured Preprocessor for MBASIC" in the January 1984 issue. In running the program on a test sample of code I discovered a problem with the routine for converting remark statements to single quote marks. On page 27 the following lines are given for the conversion:

```
1210 FOR I=1 TO N
1220        P=INSTR (L$ (I), "REM")
1230        IF P<>0 THEN L$ (I) =
            MID$ (L$ (I) ,1,P-1) +
            "'" + MID$ (L$ (I) ,P+3)
1240 NEXT I
```

This will replace the first occurrence of the string "REM" in each line of text. However, the character may appear within quoted strings or as part of a variable name. It will cause execution problems if the variables and strings are changed. Mr. Shammas' program will change the test code in Listing One (at right) into the text in Listing Two (at right).

Making a few changes to his code will intelligently change the remark statements. The speed for the conversion is about the same as Mr. Shammas'. Delete lines 1210 and 1220 and add the lines in Listing Three (at right) to the program. Listing Four (at right) shows the output from the corrected program. This should handle most programs written in MBASIC.

This code may be improved upon, however it does work. Thank you for such a fine journal.

Sincerely,
Keith L. Terrill
603 Court Street
Syracuse, New York 13208

Listing One

```
10 REM
20 REM                THIS IS A TEST FOR THE REM PROGRAM
30 REM
40 A=1 : B=2 : C=3 : REM THIS IS ANOTHER REM LINE
50 INPUT "THIS REM SHOULD NOT BE CHANGED"; A
60 PRINT "THIS REM SHOULD REM BE CHANGED" :
       REM BUT THIS ONE SHOULD
70 DATA ONE, 1, TWO, 2, REM, REM1, REM2
80 END
```

Listing Two

```
10 '
20 '                  THIS IS A TEST FOR THE REM PROGRAM
30 '
40 A=1 : B=2 : C=3 : ' THIS IS ANOTHER REM LINE
50 INPUT "THIS ' SHOULD NOT BE CHANGED"; A
60 PRINT "THIS ' SHOULD REM BE CHANGED" :
       REM BUT THIS ONE SHOULD
70 DATA ONE, 1, TWO, 2, ' , REM1, REM2
80 END
```

Listing Three

```
1209  QT$=CHR$ (34)
1210  FOR I=1 TO N
1213       P=INSTR (L$ (I) ,"REM") : IF P=0 THEN 1240
1216       P1=INSTR (L$ (I) , "DATA") : IF P1=0 THEN 1222
1219       IF P>P1 THEN 1240
1222       P2=1
1225       P1=INSTR (P2, L$ (I) ,QT$) : IF P1>P OR P1=0 THEN 1237
1228       P2=INSTR (P1+1,L$ (I),QTS) : IF P2=0 THEN 1240
1231       P2=P2+1 : IF P2<P THEN 1225
1234       P=INSTR  (P2,L$ (I) ,"REM") : IF P=0 THEN 1240 ELSE 1225
1237       L$ (I)=MID$ (L$ (I) ,1,P-1) +"'"+MID$ (L$ (I) ,P+3)
1240  NEXT I
```

Listing Four

```
10   '
20   '     THIS IS A TEST FOR THE REM PROGRAM
30   '
40   A=1 : B=2 : C=3 : 'THIS IS ANOTHER REM LINE
50   INPUT "THIS REM SHOULD NOT BE CHANGED"; A
60   PRINT "THIS REM SHOULD REM BE CHANGED" :
          'BUT THIS ONE SHOULD
70   DATA ONE, 1, TWO, 2, REM, REM1, REM2
80   END
```

Fussin' over Sal/80

Dear Doctor:

This is to express my heartfelt appreciation for the very thorough and objective review of our SAL/80 package in the February issue by Jim Kronman. I'm embarrassed to take *any* exceptions to what was written, *but* . . .

One thing I do feel obliged to remark on: Jim's example of code-density compares two printer drivers, one in naked assembler and one in SAL/80. The SAL/80 driver is 400 bytes larger than the naked assembler, which might seem to imply that SAL/80 has a run-time package of 400 bytes. This is *not* the case. SAL/80 has *no* run-time package (and *no* royalties for commercial distribution of code generated using SAL/80).

But then it might seem to imply that the code emitted by SAL/80 is 40% larger than that obtained by coding in naked assembler, which is even *worse*! I haven't seen the two drivers in question, but I am quite certain that there are differences between the two which would account for all except a maximum of about 100 bytes plus perhaps 5% of the remaining difference.

There are several UTILITIES in the packages which, all taken together, might contribute perhaps 100 bytes of excess code because of their generality. And the test/branch code can typically be squeezed by about 5% in the average case. The 5% results from the fact that jumps to jumps do occur when a forward branch is emitted in the last group of a loop-body (hence, three bytes may be squeezable), and that the HL register is saved in certain memory-reference comparisons (which gives two possibly squeezable bytes if HL was inactive at the time).

The reason I'm making such a fuss about this is that SAL/80 is the only structured assembly package I know of that actually does *optimize* the test/branch code! There is a huge decision-tree in the compiler which does all the tricks dear to the hearts of assembly-language programmers to avoid writing two conditional jumps to implement LE (<=) and GT (>) comparisions. On a branch-by-branch basis, SAL/80 emits the tightest code possible if the double-registers are to be preserved over the comparison.

Cordially,
Steve Newberry
Protools
24225 Summerhill Avenue
Los Altos, CA 94022

Ackermann Uncovered

Dear Editor,

In response to Paul Condon's question (*DDJ* No. 88, February 1984, p. 9) "Who is Ackermann and where did he publish a definition of this function?" the following:

Wilhelm Ackermann died about twenty years ago. A student of David Hilbert's, he was one of the giants in research in the Foundations of Mathematics movement of the early part of this century. The function now associated with his name first appeared in a research monograph entitled "Zum Hilbertschen Aufbau der reelen Zalen" ("On Hilbert's Reconstruction of the Real Numbers") in *Math. Annalen,* Vol. 99 (1928), pp. 118-133.

The function was used to disprove (by counter-example) a conjecture by Hilbert to the effect that the set of real numbers definable by infinite sequences of what are now called "primitive-recursive" functions were, in an intuitive sense, all the real numbers that exist. Part of the argument of supposing this to be true was another conjecture that all recursive functions are primitive-recursive. The Ackermann function is *not* a primitive-recursive function, because it can be shown to "grow faster" than any primitive-recursive function. So, by showing that there are recursive functions which are not primitive-recursive, Ackermann disproved the first conjecture. Later, in the '30s, it was shown by Alan Turing that there are real numbers which are not definable even in terms of the most broadly generalized form of recursive function.

The original formulation of Ackermann's was slightly more complex than that now used, which is:

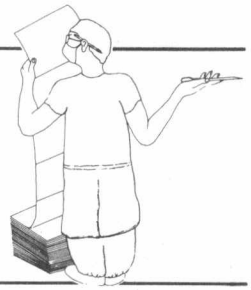
$$\begin{aligned} Ack(0, n) &= n+1 \\ Ack(m+1, 0) &= Ack(m, 1) \\ Ack(m+1, n+1) &= Ack(m, Ack(m+1,n)) \end{aligned}$$

A fuller discussion will be found in Hans Hermes' *Enumerability, Decidability, Computability,* Academic Press (1965), p. 84, and in Rozsa Peter's *Recursive Functions*, 3rd Revised Edition, Academic Press (1967), p. 106.

Didactically,
Steve Newberry
Newberry Microsystems
24225 Summerhill Ave.
Los Altos, CA 94022

DDJ

DR. DOBB'S CLINIC



by D.E. Cortesi, Resident Intern

Super Directory Fix

A while back we told how David McLanahan found that the public-domain program SD (one of the extended DIR programs) failed under CP/M Plus. Ted Shapin, of Orange, CA, writes to say that "The SD program can easily be fixed to give the correct free space under CP/M 3.0. I fixed my copy with information from D. Sternlight. At the label TMOVE there is an instruction, LDAX D. A few lines further there is a second LDAX D. After this instruction insert the line

```
ANI OFH ;DROP ARCHIVE FLAGS
```

Meanwhile, McLanahan had found another solution. A later version of SD, SD48, works just fine on CP/M Plus.

RAM-Drive Needed

Malcom Fordham, of Bowie, MD, has "an IBM PC on which I use PL/I-86 under CP/M-86. I have sufficient memory to run the compiler and linker from a RAM disk. However, so far I haven't been able to find the software to do this under CP/M-86. Do you know of such a program?"

We only know that Concurrent CP/M for the PC contains RAM-drive support out of the box. Does anyone know of RAM-drive support for plain old CP/M-86? Maybe one of the makers of add-on RAM cards supplies one; they usually supply the support code for PC-DOS.

The Great American Merchandiser

We grumbled loudly about the merchandising tactics used in our area to promote subscriptions to *PC* magazine: a come-on mailed in an envelope that emulated an official government mailing. Dave Sullivan of Palo Alto, CA, writes to say that he got a second one of those foolers. "The latest one was from *PC Tech Journal*. It appears that Ziff-Davis is the one exhibiting all the brass.

"But I have another case of gall and guts for your scrapbook," he continues. "Ziff-Davis publishes any number of magazines that touch on computers. One of the cheaper ones is called *Computers and Electronics*, and sells for $1.75 at the newsstand. The January 1984 issue has what appears to be the first of a new series of articles on assembly language programming on the PC. It looks promising, and I'll take all the help I can get to learn assembly language.

"But the last paragraph of the article reads as follows: 'If you want to pursue learning assembly language for the IBM PC, additional articles that build on this one can be read in upcoming issues of our sister publication, *PC* magazine.'

"It takes one's breath away."

What's So Hard About CP/M Plus?

We're very fond of CP/M Plus now that we have it working well. So fond that we are trying it out on other computers besides our workhorse S-100 system. Or we would be, if anybody could get it working.

There's a Tandy Model 4P, an adorable little machine, that is patiently gathering dust while it awaits the long-delayed arrival of its CP/M Plus support from Fort Worth. Tandy promised CP/M Plus when they announced the Model 4, many months back, and they still haven't delivered. They keep promising it Real Soon Now, and they keep not shipping.

Then consider our brother-in-law Bill, whom we had all primed to buy a Morrow MD-11 as his first computer, largely because it was supposed to come with CP/M Plus. Only it won't, it seems. At least the first units of the MD-11 will be shipped with plain old CP/M 2. The dealer claims "it's real hard to make CP/M Plus work with the hard disk." To which we are inclined to say, piffle, but then what do we know?

What goes on here? Bob Blum got CP/M Plus working; we got it working. David McLanahan has had it for months on his Micro Mate (a system that, he says, deserves much wider exposure than it gets). The pubs are clear and complete; the generic parts of the system are reliable; the payoff in extra function is large. So what's the holdup?

CP/M Plus, Apple, and ALS

Advanced Logic Systems is a company that managed to make CP/M Plus work quite early. We have been trying out an Apple IIe equipped with the ALS CP/M card and CP/M Plus, and it works. It isn't as nice as CP/M Plus on a *real* computer (that should bring some mail), but to a large extent that's the fault of the Apple disk drives, not the fault of ALS or the software. There just isn't a whole lot you can do with only 170K per drive.

ALS did skimp on their implementation in one area. They completely omitted implementing the CP/M Plus character I/O table and its character I/O redirection features. We can sympathize, because character I/O in the Apple is a zoo; there are so many possible permutations of serial cards, parallel cards, ROM versions, and slot numbers that it's hard to blame them for throwing up their hands. However, they didn't help the situation when they omitted documenting character I/O in any way whatsoever. The ALS bios contains some kind of support for the AUX device, for instance, but the ALS manuals contain not even a hint as to what AUX connects to.

Since our plans for the IIe involve getting MODEM to run on it so we can exchange files between it and the big system, we had to learn something about Apple I/O under the ALS bios. The ALS hot-line seems to be overloaded; they returned one of our calls out of six. The one contact we managed to make with them didn't produce any useful technical information, but it did give us the pointer we needed.

The ALS support person pointed us to a new CP/M Plus User Group being formed around the ALS card and the Apple. The group is CP/PLUG; its president is Jim Scardelis; and you can reach the group at P. O. Box 295, Little Falls, NJ 07424. Dues are $25 a year (we think, our notes are a mess) and the first quarterly newsletter was in preparation in January.

Scardelis gave us some valuable technical hints, but told us that if we wanted more we should dial into a certain RCP/M system run by one of the club members. We did and found a gold mine of Apple/ALS material. The system is the Central New York Technical RCP/M system, Mark Howard SYSOP. It is up from 1700 to 0800 hours EST, on (315) 437-4890, and is an excellent source of technical data. Thanks to Scardelis and Howard, we got our IIe running MODEM712 in fine shape.

How It's Done

Oh, you wanted to know how you do serial I/O on the Apple? Thought you'd never ask. When you are running under the ALS CP/M card, bank 1 (the bank containing the transient program area) consists of RAM chips on the ALS card itself. Global RAM starts from E000h and is also physically on the ALS card. The original Apple address space is bank 0.

That Apple address space, as seen by

(Continued to page 104)

CP/M EXCHANGE

by Robert Blum

Over the past few months it has been increasingly difficult to obtain DRI's permission to publish the application notes for CP/M. Apparently a shift in personnel at DRI had slowed the approval process, and I misunderstood how freely I could use the notes for CP/M Plus. Nonetheless, DRI has given us permission to continue publishing the application notes for CP/M V2.2 and CP/M Plus. This includes the CP/M Plus BDOS patch 02, published in the February issue of *DDJ*, which dealt with the enhancement of the LRU buffering logic of CP/M Plus. It has now been approved without changes.

When I was told of DRI's decision to allow us to continue publishing the application notes I breathed a deep sigh of relief. From the onset of our problems I was concerned that DRI was beginning to develop an attitude, or attitude problem depending on how you look at it, of pulling back from directly supporting their products. Time and again I have seen the customer support function inadequately treated by sales organizations and otherwise excellent products given undeserving tarnished reputations. I am glad that we can continue to depend on DRI to directly support their products. Starting next month we will begin publishing more application notes for CP/M.

Compatibility

Manufacturers dream of being able to move from one product generation to the next without any severe software incompatibility. And in most cases they are almost completely successful. This issue must have been a painful one at DRI while forming a product strategy for CP/M Plus. If their product marketing department is subject to the type of pressures that I have seen in similar industries they were undoubtedly presented with a difficult decision.

DRI was obligated to maintain compatibility with CP/M V2.2. It would have been incomprehensible for them to even consider breaking the tie with the hundreds of thousands of V2.2 systems currently in use. Any attempt to entice the software developers, at this time, to convert their packages to a new operating system or just to make a few modifications would have been out of the question. Being the incumbent in a time when cries are heard from every street corner that this or that new machine or software package is better than the others is worth more in advertising dollars than most companies can muster.

By having available an upward migration path the current user base would be able to easily purchase, or upgrade to, the latest version of the product. Even though not prevalent in the microcomputer industry today, we will soon find software companies demanding that customers maintain their software at current levels or pay the price of nonsupport. From the manufacturers' standpoint this is the least costly method of introducing a new product because one immediately has a qualified customer base to sell to. Support costs are reduced as well.

CP/M Plus's compatibility with older versions of CP/M is maintained at a system level. All the former BDOS calls were maintained, although in many cases extra data or status information is made available for the programmer's use. To expand the system many new calls were implemented, but again, their use is entirely up to the programmer. This is as far as DRI will go in saying that CP/M Plus is compatible with V2.2. In the System Guide they refer to CP/M Plus as simply being "upward-compatible."

In V2.2 it was common practice to "hook" into the BIOS table to enable a program to trap status codes and other information before the BDOS had a chance to act upon it. Under CP/M Plus this practice is frowned upon and is largely unnecessary due to the new BDOS error modes and the additional status information returned from each function. Programs that do make use of the old error trapping technique will continue to run without any problems provided that system alteration does not go beyond the console and list BIOS vector points. Since these were the most-used points of alteration, DRI specified that they remain available as before.

The main reason for this restriction is the address extracted from the BIOS jump table may, and probably will not, point to a real memory address. Keep in mind that in virtual memory systems (a banked CP/M Plus system falls into this category) any memory address may or may not currently reside in the processor's physical address space. Depending on the hardware manufacturer the determination of whether a memory address is real or virtual at any moment is a combined effort of both hardware and software.

In the case of CP/M Plus, some logic sections and table structures are required to be in real (common) memory at all times due to performance and memory conflict considerations. The majority of the system, however, will be virtual. When an application program makes a call to the BDOS to request a system service, a check is made to determine whether the requested portion of the system is in real memory. If not, a portion of the application program is made virtual and the missing section of the operating system is switched to occupy the real memory address space. As complicated or confusing as this may seem, it is handled quite simply in the CP/M Plus software and should not be intimidating.

To better visualize the BIOS routines that are in virtual memory refer to Listing One (page 14). All the routines pointed to by an address constant marked by a double quote are in virtual memory. You may wish to also refer back to this column in the July and August 1983 issues. In those columns I talked more about virtual memory systems and how they are generally implemented on microcomputers and particularly under CP/M Plus.

Most of the programs I have tested under CP/M Plus ran without any problems. I have found a couple of very popular programs that refused to cooperate and caused some totally unexpected and very difficult-to-find system problems until I discontinued their use. On the other hand, some programs that I would have

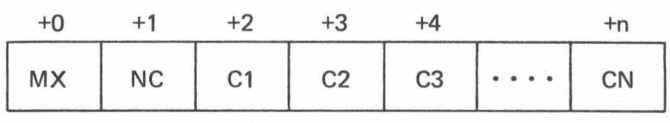


Figure
Function 10 Console Input Buffer

bet wouldn't work, do. The only suggestion that I can make to you is to ask your dealer or the manufacturer whether their programs will work under CP/M Plus. Listen to their answers very closely for key phrases such as "it should" or "I don't see why it wouldn't." If you are given an answer of this type pursue the subject further by asking whether a test was actually run and if so on what machine. I think you will find that, when backed to the wall, most of the manufacturers' representatives will admit that no direct experience is available. This is not to say that what you are looking to buy won't work; chances are it will. Just protect yourself by making sure that you can return the package if some problem does exist.

BDOS Function 10: Vastly Improved

The read console buffer function, BDOS function 10, has been vastly improved in CP/M Plus. It will now not only automatically input and edit a string from the console, but will also display the string buffer before going into edit mode.

The format of the input buffer pointed to by the DE register pair is unchanged from V2.2 (refer to Figure, page 12). The single byte, MX, is the maximum number of characters that can be placed into the buffer. And the single byte, NC, is the count of the number of characters that have been input to the buffer. The actual data that has been entered into the buffer follows the NC byte. During the data entry operation the full keyboard editing features of CP/M Plus can be used to adjust the input string.

If register pair DE is set to zero prior to calling the BDOS, function 10 assumes that an initialized input buffer is located at the current DMA address. This allows a program to put a string on the console for the user to edit. To initialize the input buffer, place the default string into the buffer following the NC byte terminated by a binary zero. When initializing the input buffer with a default string it is not necessary to set the NC byte to the number of characters in the buffer.

When a program calls the function 10 routine with an initialized buffer, function 10 operates as if the user had typed in the string. A search is made through the buffer for the binary zero terminator byte in order to count the number of active characters. This count is then placed into the NC field. The routine's attention is then tuned to allow more data to be entered from the console. At this point the user is in full keyboard editing mode and can enter more data or edit the default string. In any case hitting return will immediately send the string to the program.

As now programmed, function 10 could greatly assist in those programs that offer a default value to the user as the first choice. Refer to Listing Two (page 16) for an example of a program that makes full use of function 10's expanded capabilities.

■■J

CP/M Exchange

(Text begins on page 12)

Listing One

```
                        ; bios jump vector.

                        ; all bios routines are invoked by calling these
                        ;       entry points.

C3 0000"  ?boot:        jp    boot      ; initial entry on cold start
C3 006C'  ?wboot:       jp    wboot     ; reentry on program exit, warm start

C3 0177'  ?const:       jp    const     ; return console input status
C3 0192'  ?conin:       jp    conin     ; return console input character
C3 00DA'  ?cono:        jp    conout    ; send console output character
C3 00E6'  ?list:        jp    list      ; send list output character
C3 00E0'  ?auxo:        jp    auxout    ; send auxiliary output character
C3 0198'  ?auxi:        jp    auxin     ; return auxiliary input character

C3 006E"  ?home:        jp    home      ; set disks to logical home
C3 003F"  ?sldsk:       jp    seldsk    ; select disk drive
C3 0071"  ?sttrk:       jp    settrk    ; set disk track
C3 0077"  ?stsec:       jp    setsec    ; set disk sector
C3 007D"  ?stdma:       jp    setdma    ; set disk i/o memory address
C3 0094"  ?read:        jp    read      ; read physical block(s)
C3 00AA"  ?write:       jp    write     ; write physical block(s)

C3 0112'  ?lists:       jp    listst    ; return list device status
C3 0089"  ?sctrn:       jp    sectrn    ; translate logical to physical sector
```

```
C3 0106'  ?conos:     jp    conost    ; return console output status
C3 017D'  ?auxis:     jp    auxist    ; return aux input status
C3 010C'  ?auxos:     jp    auxost    ; return aux output status
C3 00D2'  ?dvtbl:     jp    devtbl    ; return address of device def table
C3 0000*  ?devin:     jp    ?cinit    ; change baud rate of device

C3 00D6'  ?drtbl:     jp    getdrv    ; return address of disk drive table
C3 00CB"  ?mltio:     jp    multio    ; set multiple record count for disk i/o
C3 00CF"  ?flush:     jp    flush     ; flush bios maintained disk caching

C3 0000*  ?mov:       jp    ?move     ; block move memory to memory
C3 0000*  ?tim:       jp    ?time     ; signal time and date operation
C3 0250'  ?bnksl:     jp    bnksel    ; select bank for code execution and default dma
C3 0085"  ?stbnk:     jp    setbnk    ; select different bank for disk i/o dma operations.
C3 0000*  ?xmov:      jp    ?xmove    ; set source and destination banks
```

End Listing One

Listing Two

```
              .z80

bdos          equ   5             ;BDOS function call entry point

set_dma       equ   26            ;set DMA address function code
read_buffer   equ   10            ;read/edit console buffer function code

              ld    sp,stack      ;set a local stack

              ld    c,set_dma     ;set DMA address to input buffer
              ld    de,buffer     ;*
              call  bdos          ;*

              ld    c,read_buffer ;edit string in buffer
              ld    de,0          ;*
              call  bdos          ;*

              rst   0             ;return to CP/M

buffer        equ   $
              defb  10            ;maximum number of characters allowed
              defb  5             ;number of active characters
              defb  'string',0    ;initial string - zero delimited
              defb  '     '       ;room for maximum input length

              defs  30
stack         equ   $

              end
```

End Listings

Optimizing Strings in C

by Edward McDermott

Edward McDermott, 12 Manor Haven Rd., Toronto, Ontario, Canada M6A2H9.

The C programming language offers many advantages: flexibility, speed, and portability. Routines that could take months in assembler can be written and tested much more easily in C. It supports true recursion and produces assembler language source, which allows users to optimize portions of their code.

Optimization is the process of improving the efficiency of a program; i.e., getting more bang for your buck, whether the buck is time or memory. Basically, you try to make the program smaller, faster, or both, although you sometimes have to trade size for speed or vice versa. Optimization can apply to one particular program or to a group of programs. One problem with optimization is that it takes both time and effort.

The following optimization approaches have certain advantages and disadvantages:

Improving the compiler output by improving the compiler

Improving the compiler output by tinkering

Improving the individual program's algorithm

Rewriting entire routines in assembler

To improve the compiler output, you must have the source for the compiler and must know enough to improve the code generation logic. If you have those skills, please try it and publish the results. For the rest of us mere mortals, however, this is a foolhardy exercise that can bring disaster. Besides, it takes a lot of time.

C programmers have another alternative: to recognize a commonly used pattern of code that can be improved and to write a program that tinkers with the compiler's output to improve that part of the code. This approach is very similar to the optimization portion of the Small-C compiler.

Improving an individual program by rewriting all or part of it is a perfectly valid approach, except that the payback doesn't go any farther. The rest of your programs are still the same.

The final approach is write some commonly used routines in assembler. This will speed things up and you can generalize this approach to all your programs by recompiling them. It improves both execution speed and size (to some extent). The problem is that these routines are no longer portable from machine to machine.

If you choose this last approach, its success depends on which functions you select for optimization. The ideal candidates would have the following characteristics:

Contains tight loops

Is used by almost every program

Is small enough to easily rework

C is an ideal language for editors, compilers, operating systems and so on; that was what C was designed for. Although every one of these applications involves a good deal of string handling, C has no string manipulation verbs. All string handling is done on a character-by-character basis, often by very small functions, which contain some very tight loops.

Another aspect to consider is any special commands of the CPU you are using. Can any functions built into the chip be taken advantage of by another means? The Z80 chip has a few commands that are unavailable on the 8080, specifically: LDIR, LDDR, LDI, LDD, CPIR, CPDR, CPI, and CPD. Each of these instructions is the equivalent of a number of 8080 instructions and, properly used, can be substituted for a number of instructions in C.

For example, the LDIR command takes the contents of the memory position pointed at by the HL register pair, transfers those contents to the memory position specified by the DE register pair, increments HL and DE, and decrements BC until the BC register pair is zero. Perhaps this makes it a little clearer:

```
while (--BC) *DE++ = *HL++;
```

Does that seem familiar? It is, in fact, almost identical to the STRCPY function in *The C Programming Language* by Kernighan and Ritchie. The principal difference lies in the delimitation of the loop.

If you run on a Z80 chip, string manipulation is a natural choice. But the question that haunts all optimization efforts still remains: Is the payback worth the effort? To try and answer that question we chose to optimize the following four functions:

STRCPY – copy a string from one place to another

STRINIT – initialize a string to a specified value

STRLEN – find the length of a string

CMATCH – find a character within a string

Since the Small-C compiler produces assembler code, our first approach was to review its output. In small loops the compiler produces code that does not optimally use the registers, so we completely rewrote the functions; most of the speed gain can be attributed to this rewrite. This optimization should apply to 8080-based machines as well.

The accompanying Listing (page 20) gives the new routines in Z80 code. We chose Z80 both to take advantage of its special OP codes and to emphasize the major disadvantage of this type of optimization: the loss of portability. Readers, however, can adapt these routines to the 8080 environment fairly easily, for use with C or other languages.

Usually you judge the success of an optimization on the following criteria:

Is it faster and if so how much?

Is it smaller and if so how much?

Does it suggest other methods of improvement?

Was it worth the time and effort to do?

Faster

We acquired the timings here by executing each command 10,000 times on a string of 30 characters. The first set of timings is for the driver program, without any string manipulation at all, to see the impact of the code used to handle the test. The second set is using C routines, compiled under the Small-C compiler, version 2. The third set of timings is for the same program but using the assembler routines. The timings are as follows:

Null test	3 sec.
Small C (v2)	195 sec.
Assembler	16 sec.

We also compared the CMATCH routine to the rest, since this routine is significantly longer and more involved in both languages:

Null test	1 sec.
Small C (v2)	90 sec.
Assembler	6 sec.

If you remove the constant load of the code to drive the test program, you

find the new routines are 14.8 times faster. Skeptics can ignore the impact of the loops and still end up with new routines that are 12.2 times faster.

Smaller

The memory saving were illustrated by the following:

Language	Size
Small C(v2)	254 bytes
Assembler	116 bytes

The assembler code for the four routines was less than half the size of the comparable C code. However, saving 138 bytes is not going to make anyone jump for joy. With fine tuning, this figure could be improved, but once again it is a matter of cost versus gain.

Other Improvements

Our first thought was to optimize other small routines for handling strings. But part of the savings had come from using all three register pairs and keeping the values in the registers throughout the loop. Where the number of parameters exceeds three (one for each of BC, DE, and HL register pairs), the potential savings drop and the effort begins to rise.

Another improvement would be using these optimized routines as heavily as possible. That has some major implications for the style of C code a programmer uses. You would have to build up your functions out of smaller optimized functions instead of writing complex character-by-character processes.

Some other potential candidates for optimization are as follows:

Convert to upper case and convert to lower case

Find string

Compare string

Extract part of a string

All string manipulation functions should not necessarily be optimized. The gain in speed through improving screen and printer routines is hidden by the limited transmission speed to these devices for most small systems; you won't see any improvement. Most likely your disk I/O routines are already in assembler to interface the language to your particular operating system.

By the way, the STRCPY and STRINIT functions were designed to return the address of the resulting string. This may seem an unnecessary complication, but it does allow the programmer to nest functions within each other when writing the C code.

Time and Effort

Whether the optimization was worth the time and effort depends on how you value your time and on your requirements for speed and/or memory. My personal expectation for speed improvement was somewhere in the order of five times faster. The improvement, however, was between 12 and 15 times faster. Since creating these routines, testing them, correcting them, and correcting them again resulted in more speed than I expected, the optimization proved successful. You have to decide if it's worth your time to type these routines in and use them in your code.

The improvement you can achieve by converting small routines into assembler is significant, if not startling. Leaving the original C routine as a comment in the assembler code lets you combine the best of both worlds: the most efficient routine and the original portable routine. Any small, tight looping function that is concerned with three or fewer variables and is commonly used is an ideal candidate. However, to get the most out of such an optimization, you must go on to build more complex functions out of these smaller ones. ■■J

(Listing begins on page 20)

Editor's note on some enhancements: Any comparison of the accumulator to 0 can be replaced with an OR A command, and any initialization of the accumulator can be replaced with an XOR A command. This improves speed and reduces memory requirements.

Some places where one could make the code tighter are perhaps less obvious. In lines 50-54 of the listing, the following command sequence could replace the existing ones:

```
LD   A,B
OR   B
JR   Z,STRIN2
```

Likewise, lines 97-101 could be replaced with

```
LD   A,B
OR   C
JR   Z,CMA1
```

and lines 103-106 could be replaced with

```
LD   A,B
OR   C
JR   NZ,CMAX1
```

In lines 104 and 106, the author used JP instructions instead of JRs. While JR is probably the better instruction, JP is safer in the case of the author's assembler and if one is uncertain about the distance to the branch location. In this case, JR should work quite well.

Optimizing Strings in C (Text begins on page 18)

```
 1: #asm
 2: ;/*
 3: .Z80
 4:
 5: ;strcpy(s,t) char *s, *t;
 6: ;       { int ret; ret = *s; while (*s++ = *t++); return(ret)  ; }
 7:
 8: STRCPY::
 9:         POP     BC              ; pop return address
10:         POP     HL              ; pop address of t
11:         POP     DE              ; pop address of s
12:         PUSH    DE              ; restore stack
13:         PUSH    HL
14:         PUSH    BC
15:         PUSH    DE              ; save address of start of s
16:
17: STRLP:  LD      A,(HL)          ; transfer loop
18:         CP      00              ; test for zero in t
19:         JR      Z,STREXT        ; if so exit
20:         LDI                     ; move incrementing
21:                                 ; s++ = t++
22:         JR      STRLP           ; continue loop
23:
24: STREXT: LD      A,00            ; zero final bite in s
25:         LD      (DE),A
26:         POP     HL              ; return original address of s
27:         RET
28:
29: ;strinit(s,c,i ) char *s; char c ; int i ;
30: ;       { int ret; ret = *s;
31: ;       while (i-- ) *s++=c;
32: ;       return(ret) ; }
33:
34: STRINIT::
35:         POP     IX              ; pop return address
36:         POP     BC              ; pop i  (len for init)
37:         POP     DE              ; pop c  ( init character
38:         POP     HL              ; POP ADDRESS OF STRING
39:         PUSH    HL              ; restore stack
40:         PUSH    DE
41:         PUSH    BC
42:         PUSH    IX
43:         PUSH    HL              ; save address of start of s
44:         LD      A,E             ; LOAD A  with c character
45:         LD      (HL),A          ; store character in begin of s
46:         POP     DE              ; SET DE to begin of s
47:         PUSH    DE
48:         INC     DE              ; point DE to next bye
49:         DEC     BC              ; REDUCE FOR CHAR TAKEN
50:         LD      A,00
51:         CP      C             ; MAKE SURE BC > 00
```

```
52:          JR       NZ,STRIN1
53:          CP       B
54:          JR       Z,STRIN2
55:
56: STRIN1:  LDIR                          ; propogate it through string
57: STRIN2:  POP      HL                   ; get address of s to return
58:          RET
59: ;/*  strlen -- return length of string s (page 98) */
60: ;strlen(s) char *s;
61: ;       { int p ;  p =s;
62: ;         while (*s) ++s  ;
63: ;         return (s-p);}
64:
65: STRLEN::
66:          POP      BC                   ; pop return address
67:          POP      HL                   ; pop address of s
68:          PUSH     HL                   ; restore stack
69:          PUSH     BC
70:          PUSH     HL                   ; save address of start of s
71:          LD       BC,0FFFFH            ; BC IS BYTE COUNT (decremented)
72:          LD       A,00                 ; searching for a  00
73:          CPIR                          ; equivalent of while (HL++)
74:          POP      DE                   ; DE = s
75:          SBC      HL,DE                ; HL = HL - DE
76:          DEC      HL                   ; correction for count last char
77:          RET                           ; RETURN (HL);
78:
79: ;cmatch(s,p,i )                              /* find first p  in str s */
80: ;         char s[] ; int i, p ;
81: ;         { if (i > strlen(s)) return(0);
82: ;         while (s[i] != 0 )
83: ;                 { if (p == s[i++]) return  (i); }
84: ;         return 0; }
85:
86: CMATCH::
87:          POP      IX                   ; pop return address
88:          POP      BC                   ; pop i  (indent for init)
89:          POP      DE                   ; pop p  (search character
90:          POP      HL                   ; POP ADDRESS OF STRING s
91:          PUSH     HL                   ; restore stack
92:          PUSH     DE
93:          PUSH     BC
94:          PUSH     IX
95:
96:          PUSH     BC                   ; save BC for count of bytes
97:          LD       A,0                  ; LOAD A  with 0
98:          CP       B                    ; IF BC == 0  THEN GO TO CMA1
99:          JR       NZ,CMA0              ;
100:         CP       C
101:         JR       Z,CMA1
102: CMA0:   CPIR                          ; check 00  before end of BC
103:         CP       B                    ; IF (BC != 0 ) return 0
104:         JP       NZ,CMAX1
105:         CP       C
```

```
106:          JP        NZ,CMAX1
107:                                    ; else continue
108: CMA1:    POP       BC              ; restor BC = i for offest count
109:                                    ; i is within  the string ?
110: CMA2:    INC       BC             ; count bytes
111:          LD        A,(HL)
112:          CP        00              ; end of string?
113:          JR        Z,CMAX
114:          CP        E               ; check search character
115:          JR        Z,CMAE         ; if (HL = p) contin
116:          INC       HL             ; HL ++
117:          JR        CMA2
118:
119: CMAE:    LD        H,B            ; return (HL = BC)
120:          LD        L,C
121:          RET
122:
123:          ; i is beyond the string end
124: CMAX1:   POP       BC               ; restore stack
125: CMAX:    LD        HL,00H           ;RETURN (NULL);
126:          RET
127: #endasm
```

End Listing

Expert Systems and the Weather

by Jack Park

Somebody famous once said: "This is gonna get me in trouble." Since I am about to discuss a program that is capable of predicting the weather (sometimes), I figure I'll get in trouble, but here goes.

I chose predicting the weather as the subject of an early effort at knowledge engineering using EXPERT-2, a tiny fifth-generation type language.[1] My friends in the business of weather prediction think this is somewhat amusing, but the results are quite interesting and possibly useful. I deliberately kept the knowledge engineering exercise simple since I intended to run this program in a 48K Apple II, using a Forth system with EXPERT-2 on top and the application program, the weather predictor.

Expert systems are computer programs that are specially designed to perform inferences: to prove things. EXPERT-2 is a program that allows individuals to write task programs much like BASIC lets users write programs. It is a high-level language written for performing logic inferences using rules written into the user's program. These rules may be used to solve important problems, like predicting the stock market . . . or the weather.

EXPERT-2 allows two methods for encoding knowledge and information into the task program – in this case, the weather predictor:

IF-THEN statements

Analytic subroutines

IF-THEN statements, or production rules, allow the user to direct the inference process by entering into the computer specific statements that say, "if something is true, then something else is true." The EXPERT-2 syntax is an extension of the IF-THEN syntax used to develop an animals game.[2,3] The extensions include allowing statements to be entered in a negative context (IFNOT, ANDNOT) and calling analytic subroutines.

Analytic subroutines were added to give the programmer access to the full power of the underlying Forth system. If EXPERT-2 had been written in, say, Pascal, then the analytic subroutine calls (IFRUN, IFNOTRUN, ANDRUN, ANDNOTRUN) would permit access to that environment. As it is, EXPERT-2 was written in Forth, primarily because that's the program environment in which I work best. Users of EXPERT-2, however, need not know or understand Forth; if analytic subroutines are not needed, no Forth coding is required.

The weather predictor presented here does use analytic subroutines. These routines are used in two ways:

To prompt the user to input data

To process data and return truth values to the rules

The IF-THEN rules call the appropriate analytic subroutines when answers about weather data are needed.

The IF-THEN rules contain knowledge about the weather. This knowledge is encoded as antecedents (e.g., IF barometric pressure is falling rapidly) and consequents (e.g., THEN the weather is turning bad). Consequents follow appropriate antecedents. Designing rules to correctly encode knowledge is called knowledge engineering.

Knowledge Engineering

Someone once asked me if EXPERT-2 ever adds to its own knowledge base or makes original statements. As it turns out, it often makes original statements – especially during system debugging. Early versions of EXPERT-2, in fact, were self-modifying (not by design!). Test runs were incredibly mystifying. Knowledge engineering, as I use the term here, assumes one has available a working inference machine: a program that permits inferences to be performed on a set of rules. EXPERT-2 is one of the early microcomputer-based inference machines available; I expect there eventually will be a slew of them.

Knowledge engineering is the process of defining a problem to solve (something for the expert to do, especially something useful), encoding into a program the expertise required to solve the problem, testing, and finally validating the expert program.

To encode knowledge, one needs an expert whose brains (expertise) are accessible for encoding. As it turns out, that's not a generally available commodity; experts may be around, but it takes charm, wit, and experience to get at their brains. Trying to determine what tools an expert brings to bear on a problem often gets you a textbook set of answers. Textbook answers are not necessarily what you need to solve a problem. Factors including years of experience, intuition, and a host of other issues must be explored before a knowledge engineer fully exhausts all the elements needed to solve a particular problem.

For the weather predictor, I chose to explore only books and magazines, partly because I didn't want to call in any chits I may have with the weather service and partly because I discovered some interesting prospects in the literature for homebrew weather prediction that could be translated to the EXPERT-2 system. A set of programs have been presented for computers that run BASIC.[4] These programs ask certain questions and issue weather predictions based on the answers. Because the algorithm used in the programs relates well to the general literature on weather prediction, I selected it as a representative "expert" set of rules.

For additions and changes to the rules derived from the magazine article I drew on the Heathkit Weather Training Manual, which has a section on weather forecasting.[5] Selected rules on cloud motion, for example, were used to enhance the rule base.

Analytic subroutines are then used to acquire weather data from the user and to process that data as the inference machine needs answers.

Weather patterns are quite sensitive to local terrain influences. Therefore, the responses the weather predictor offers may not be appropriate to, say, a canyon region where you live. It seems to work well where I live. I leave it as an exercise for the reader to validate the weather predictor program in his or her own area.

System Operation

Rules are typed into a disk file using the editor available with the underlying Forth system. EXPERT-2 is then loaded and compiled on top of Forth as a task. Finally, the user task (in this case, the weather predictor) is loaded on top of EXPERT-2. First Forth compiles the analytic subroutines and any variables or constants. EXPERT-2 then compiles the IF-THEN rules once the word RULES is encountered. Rule compiling ends with the word DONE.

Forth, in its usual fashion, tells you that compiling has ended by printing OK on the screen. At this time it should be

Jack Park, Helion Inc., Box 445, Brownsville, CA 95919.

safe to type RUN and start the forecasting program. RUN initializes the data base by asking appropriate questions then calls EXPERT-2 to solve the problems. The problem is to prove one of four possible hypotheses, identified in the rule base by THENHYP. The four hypotheses are:

Weather is OK

Weather is improving

Weather is turning bad

Insufficient data to perform a forecast

EXPERT-2 takes the first hypothesis and begins the task of proving it. The routine that performs this task is DIAGNOSE, which is called by RUN. DIAGNOSE selects the hypothesis and passes it deeper into the EXPERT-2 program. EXPERT-2 collects all the rules that might support a proof.

If the hypothesis is "weather is OK," then all rules that have as one of their consequent fields "weather is OK" are tagged and drawn into the inference process. One such rule is:

```
IFRUN BP>30.2
ANDRUN BP-SLOW-FALL
ANDRUN WDIR
BECAUSE WINDS FROM SW, W,
    OR NW
THENHYP WEATHER OK
ANDTHEN FAIR AND WARM
    NEXT 48 HRS
```

One of the consequent fields (in this case, the first consequent) is the same as our hypothesis; in fact, this rule happens to be the one selected to define an hypothesis with THENHYP. EXPERT-2 finds this rule and tries to prove it.

The inference machine takes each of the antecedents, one at a time, and tests them. If all pass with a truth value of TRUE, the consequents are taken as true, and the proof is complete. If any antecedent fails with a truth value of FALSE, the rest of that rule is not tested. Anything that is "learned" along the way, however, is saved as a fact. If the system asks you which way the clouds are moving, for example, your answer becomes a fact so that EXPERT-2 does not have to ask you that question again.

If EXPERT-2 proves a rule, DIAGNOSE tells you about it, and the program ends with I CONCLUDE . . . . If, on the other hand, a given hypothesis cannot be proven, then the next available hypothesis is selected, and a new proof is started. The new proof has available to it everything that the system learned during all previous proof attempts (all attempts, that is, since RUN was typed). If no proof is available on any hypothesis, EXPERT-2 typically states, CANNOT PROVE ANYTHING.

The Weather Predictor

You may wonder how such a simple-looking program can do any useful weather prediction; most people ask about this. As it turns out, the program concerns itself only with large-scale frontal movements. These fronts are best visualized (but not correctly described) as large bubbles of air hundreds of miles across, sliding around over the ground, bumping into each other, and generally moving from the west toward the east. Some of these bubbles are huge low-pressure masses of air, and some are large high-pressure masses of air.

When a low-pressure mass of air approaches your area, the barometer will spot the frontal motion and indicate that the barometric pressure is falling. As that air mass leaves your area, the barometer will begin to rise. Thus, by using readings of barometric pressure and asking questions about which way the winds are blowing, the weather predictor attempts to determine which of its "known" frontal-air-mass patterns best describes the conditions present in the atmosphere. If it finds a pattern that fits, it issues a prediction. The patterns it looks for, however, must be calibrated for your area.

As a simple exercise in knowledge engineering, the weather predictor demonstrates nearly the full capabilities of EXPERT-2. The program uses information on barometric pressure, surface wind direction, and upper air wind direction (as evidenced in upper cloud motion).

The rules begin with WALL, a word that lets you clear this rule base from memory if you want to load another program (by typing FORGET WALL). Following WALL, all analytic subroutines are loaded, written in Forth. This version (Listing One, page 27) is written in an extended Forth-79 dialect and based on the 40-character line width of a standard Apple II. Readers who type this in on other systems are encouraged to explore the enormous advantages of larger line widths; some awkward statements often are created when one is constrained to a narrow screen.

Following the analytic subroutines, RULES starts EXPERT-2's rule compiler. This invokes the IF-THEN syntax that the listing illustrates.

This weather predictor tests for 19 different possible combinations of weather conditions, using 19 basic weather prediction rules. If the requested weather information you type in fits within one of the 19 possible data combinations, a prediction will be issued. If the data falls outside all possible tests, the system will issue the statement INSUFFICIENT DATA FOR A FORECAST.

Since EXPERT-2 would rather say CANNOT PROVE ANYTHING, the last rule in the set stops that statement and issues a custom proof-collapse statement. That way, if EXPERT-2 is unable to prove any of the three main hypotheses, it is offered the last hypothesis as a consolation prize. This allows you to end an unsuccessful session with any statement you wish.

Three sets of rules are available for checking on upper level clouds. These rules, which support some of the 19 main rules by asking special questions about such clouds, underscore one of the weaknesses of EXPERT-2: its inability to handle an "I don't know" answer. That answer, in fact, does not exist in the syntax. An astute knowledge engineer, however, is still burdened with the likely event that this particular answer will be needed. Weather predictor handles the "don't know" case by allowing you to answer N for no if either you cannot see the upper clouds (there may not be any clouds to see) or you haven't gone outside to look as instructed. If you answer no, the system assumes the cloud motion supports whatever rule it happens to be trying to prove at the time.

This derives from the second rule in each of the three rule groups. The second rule, in effect, says: After dealing with the instructions (MESSAGE1), IFNOT you can see the clouds THEN clouds indicate . . . . One of these exists for each of the three possible conditions. A true condition is forced if you answer no. I am quite sure I'm gonna get in trouble for this.

A typical terminal session with this weather predictor follows:

1. System starts by asking barometric pressure; you enter 30.2.
2. System next asks what the barometric pressure is doing; you enter the code for falling slowly.
3. System asks for the direction from which the wind blows; you enter the code for south.
4. The terminal response is:

```
NOW, WE CAN PREDICT THE
WEATHER.
I MIGHT BE ASKING A FEW MORE
QUESTIONS.
I DEDUCE
    RAIN WITHIN 24 HRS
I DEDUCE
    WEATHER TURNING BAD
I CONCLUDE
    WEATHER TURNING BAD
OK (Forth's usual prompt)
```

No further questions were asked in that session. This particular session included data that triggered (successfully) the following rule:

```
IFRUN BP>30.1
    (30.2)
ANDRUN BP-SLOW-FALL
    (pressure falling slowly)
ANDRUN SDIR
    (southerly direction winds)
BECAUSE WINDS FROM SE OR S
THEN RAIN WITHIN 24 HRS
```

ANDTHEN WEATHER TURNING BAD

This rule illustrates a number of the design methods that were used in this knowledge engineering exercise. First, the final consequent, "weather turning bad," helps direct the proof search. If you read the rules, especially the consequent fields, you will see that a search could be conducted on dozens of different possible consequents: "rain within 24 hrs," for example. If I had set as many hypotheses as there are different consequents in this program, the hypothesis field would have been enormous. Instead I chose three primary consequents as "search pruners" to limit the search requirement. All possible rules that have a particular hypothesis are collected at the same time for proof. The first rule to pass the test terminates all further tests. Other than that, "weather turning bad" is a pretty useless consequent.

A BECAUSE field is included in this rule to allow the programmer to tack explanations in where the code might otherwise be incomprehensible. For example, if the name chosen for an analytic subroutine that is called to check wind direction is not particularly descriptive, a BECAUSE statement is added. If the system asks a question in the form IS THIS TRUE?, you can answer Y for yes, N for no, or W for "Why did you ask me that question?" It allows you to get an explanation, which is where the BECAUSE clause is handy.

The BECAUSE clause is also handy when passing around source code for programs. One of the advantages of the EXPERT-2 style program is that it is largely self-commenting. Where it is not self-commenting, you can add BECAUSE clauses.

This EXPERT-2 program's heavy use of analytic subroutines reduces the number of questions the program might be inclined to ask at the terminal. By contrast, the animals game mentioned earlier uses absolutely no analytic subroutines: whatever it learns in the proof process is gained from the terminal. Expert programs are much more interesting if they ask something at the terminal.

Going Beyond

EXPERT-2 is a high-level language for experimenters in early fifth-generation machines. It is made available in source code form, loadable on a variety of Forth systems. Using the source, this weather predictor program, and a lot of sweat equity, I expect some pretty strong weather predictor programs will emerge.

Weather prediction is largely a data processing and pattern recognition effort, one well suited to an expert system. By adding some simplified models related to atmospheric physics and coupling the program (through analytic subroutine functions) to an automatic weather station, you could perhaps create a new program for predicting weather hazards for farmers or sporting events.

Weather forecasters rate each other by a value they call "skill." Flipping a coin usually gets a skill of 0.5 or 50% right. It will be interesting to see what skill this program and its derivatives achieve. Since local weather prediction is sensitive to local geographic terrain effects, this program will need some fiddling to raise its skill, perhaps even to the 50% level.

Some thought might go into letting the program run on a pseudo-real-time basis, keeping track of its own skill. It also might be written to fiddle with its own inferences in an attempt to raise its skill: a self-learning system. You will have to make sure, however, that it does not learn how to fiddle with its skill algorithm.

■■J

(Continued on page 93)

Weather Predictor

(Text begins on page 24)

```
Screen 1
( WEATHER PREDICTOR - EXPERT-2        JP)
: WALL ; FORTH
VARIABLE MESS1          ( toggle flag )
2VARIABLE BP         ( barometric press)
VARIABLE DIR 2 ALLOT       ( dir code )
VARIABLE BP?    ( pressure trend code )

: RUN 0 MESS1 ! DIR 4 ERASE HOME CR ."
     NIMBLE WEATHER PROGRAM" CR CR  ."
THIS PROGRAM ATTEMPTS TO PROVE ONE OF
THE FOLLOWING : " CR                ."
  THE WEATHER IS OK" CR             ."
  THE WEATHER IS IMPROVING" CR      ."
  THE WEATHER IS DETERIORATING" CR  ."
NOT ALL WEATHER CASES ARE INCLUDED IN
THE FORECAST ALGORITHM." CR CR      ."
BEFORE WE ATTEMPT A FORECAST,
I NEED SOME DATA." CR CR            ."
WHAT IS THE BAROMETRIC PRESSURE READING?
< EXAMPLE - TYPE 30.2 > " INPUTD# CR CR
  BP 2!
-->
  CONTINUED ON NEXT SCREEN

Screen 2
( WEATHER PREDICTOR - EXPERT-2        JP)
                                     ."
SELECT FROM THE FOLLOWING : " CR     ."
 1 - BP IS STEADY" CR                ."
 2 - BP IS RISING SLOWLY" CR         ."
 3 - BP IS RISING RAPIDLY" CR        ."
 4 - BP IS FALLING SLOWLY" CR        ."
 5 - BP IS FALLING RAPIDLY " INPUT#
   BP? ! CR                          ."
THE WIND IS BLOWING FROM WHICH OF THE
FOLLOWING DIRECTIONS ? " CR          ."
< N, NE, E, SE, S SW, W, NW > "
   DIR 3 EXPECT CR CR                ."
NOW WE CAN PREDICT THE WEATHER." CR ."
I MIGHT BE ASKING A FEW MORE QUESTIONS"
   DIAGNOSE ( start EXPERT-2 ) ;
-->

Screen 3
( WEATHER PREDICTOR - EXPERT-2        JP)
( analytical subroutines follow )
: BP>30.2 ( -- TF )
   BP 2@ 30.2 D> ;

: BP>30.1 ( -- TF )
   BP 2@ 30.1 D> ;

: BP<30.1 ( -- TF )
   BP 2@ 30.1 D< ;

: BP>29.9 ( -- TF )
   BP 2@ 2DUP 29.8 D< >R 29.8 D= R> OR ;
HEX ( now set-up direction tests )
: WDIR ( -- TF )
   DIR @ 5753 ( SW ) OVER = SWAP 57 ( W )
   OVER = SWAP 574E ( NW ) = OR OR ;
: SDIR ( -- TF )
   DIR @ 4553 ( SE ) OVER = SWAP 53 ( S )
   = OR ;
-->
```

(Continued on next page)

Weather Predictor (Listing Continued, text begins on page 24)

```
Screen 4
( WEATHER PREDICTOR - EXPERT-2          JP)
: NEDIR DIR @ 454E ( NE ) OVER = SWAP
  45 ( E ) = OR ;
: NDIR NEDIR DIR @ 4E ( N ) = OR ;
: EDIR NEDIR DIR @ 4553 ( SE ) = OR ;
: SEDIR SDIR DIR @ 45 ( E ) = OR ;
: SWDIR DIR @ 53 ( S ) OVER = SWAP
  5753 ( SW ) = OR ;
: BP-SLOW-FALL BP? @ 4 = ;
: BP-RAPID-FALL BP? @ 5 = ;
: BP-SLOW-RISE BP? @ 2 = ;
: BP-RAPID-RISE BP? @ 3 = ;
: BP-STEADY BP? @ 1 = ;
DECIMAL
-->

Screen 5
( WEATHER PREDICTOR - EXPERT-2          JP)

: MESSAGE1 MESS1 @ NOT
  IF CR                                 ."
FOR THE NEXT QUESTION, YOU SHOULD STAND
OUTSIDE WITH YOUR BACK TO THE SURFACE
WIND." CR                               ."
YOU NOW MUST OBSERVE THE DIRECTION UPPER
LEVEL CLOUDS ARE MOVING." CR            ."
YOU OBSERVE THEM TO BE MOVING FROM YOUR
RIGHT, FROM YOUR LEFT, OR IN A DIRECTION
PARALLEL TO THAT WHICH YOU ARE FACING."
     CR                                 ."
USE THIS INFORMATION TO ANSWER THE
FOLLOWING QUESTION. IF YOU ARE UNABLE TO
SEE THE UPPER LEVEL CLOUDS, ANSWER NO TO
THE FOLLOWING QUESTION" CR
  THEN 1 MESS1 ! TRUE ;
-->

Screen 6
( WEATHER PREDICTOR - EXPERT-2          JP)
RULES  ( start the rule compiler )
( deteriorate test using clouds )
 IFRUN MESSAGE1
 ANDIF YOU CAN SEE UPPER LEVEL CLOUDS
 ANDIF CLOUDS MOVING FROM YOUR LEFT
THEN CLOUDS INDICATE BAD WEATHER
 IFRUN MESSAGE1
 ANDNOT YOU CAN SEE UPPER LEVEL CLOUDS
THEN CLOUDS INDICATE BAD WEATHER
( improving test using clouds )
 IFRUN MESSAGE1
 ANDIF YOU CAN SEE UPPER LEVEL CLOUDS
 ANDIF CLOUDS MOVING FROM YOUR RIGHT
THEN CLOUDS INDICATE WEATHER IMPROVING
 IFRUN MESSAGE1
 ANDNOT YOU CAN SEE UPPER LEVEL CLOUDS
THEN CLOUDS INDICATE WEATHER IMPROVING
-->

Screen 7
( WEATHER PREDICTOR - EXPERT-2          JP)
( steady test using clouds )
 IFRUN MESSAGE1
 ANDIF YOU CAN SEE UPPER LEVEL CLOUDS
 AND CLOUDS MOVE PARALLEL TO DIRECTION
THEN CLOUDS INDICATE STEADY WEATHER
 IFRUN MESSAGE1
 ANDNOT YOU CAN SEE UPPER LEVEL CLOUDS
THEN CLOUDS INDICATE STEADY WEATHER
-->

Screen 8
( WEATHER PREDICTOR - EXPERT-2          JP)
 IFRUN BP>30.2
 ANDRUN BP-SLOW-FALL
 ANDRUN WDIR
 BECAUSE WINDS FROM SW, W, OR NW
THENHYP WEATHER OK
ANDTHEN FAIR AND WARM NEXT 48 HRS
 IFRUN BP>30.2
 ANDRUN BP-STEADY
 ANDRUN WDIR
 BECAUSE WINDS FROM SW, W, OR NW
THEN CONTINUED FAIR
ANDTHEN LITTLE TEMPERATURE CHANGE
ANDTHEN WEATHER OK
 IFRUN BP>30.1
 ANDRUN BP-STEADY
 ANDRUN WDIR
 BECAUSE WINDS FROM SW, W, OR NW
THEN FAIR WEATHER
ANDTHEN LITTLE CHANGE NEXT 48 HRS
ANDTHEN WEATHER OK
-->

Screen 9
( WEATHER PREDICTOR - EXPERT-2          JP)
 IFRUN BP>30.1
 ANDRUN BP-SLOW-FALL
 ANDRUN NEDIR
 BECAUSE WINDS FROM NE OR E
 AND SEASON IS SUMMER
 BECAUSE NOT WINTER
 AND CLOUDS INDICATE STEADY WEATHER
THEN RAIN MAY NOT FALL
ANDTHEN STEADY FOR SEVERAL DAYS
ANDTHEN WEATHER OK
-->

Screen 10
( WEATHER PREDICTOR - EXPERT-2          JP)
 IFRUN BP<29.8
 ANDRUN BP-RAPID-RISE
THENHYP WEATHER IMPROVING
ANDTHEN CLEARING AND COLDER
 IFRUN BP<30.1
 ANDRUN BP-SLOW-RISE
 ANDRUN SWDIR
 BECAUSE WINDS FROM S OR SW
 AND CLOUDS INDICATE WEATHER IMPROVING
THEN CLEARING WITHIN A FEW HRS
ANDTHEN FAIR NEXT SEVERAL DAYS
ANDTHEN WEATHER IMPROVING
-->

Screen 11
( WEATHER PREDICTOR - EXPERT-2          JP)
 IFRUN BP>30.1
 ANDRUN BP-SLOW-FALL
 ANDRUN EDIR
```

```
 BECAUSE WINDS FROM NE, E, OR SE
 AND CLOUDS INDICATE BAD WEATHER
THEN RAIN IN 12 - 18 HRS
ANDTHEN WEATHER TURNING BAD
 IFRUN BP>30.1
 ANDRUN BP-RAPID-FALL
 ANDRUN EDIR
 BECAUSE WINDS FROM NE, E, OR SE
THEN WINDY, RAIN WITHIN 12 HRS
ANDTHEN WEATHER TURNING BAD
-->


Screen 12
( WEATHER PREDICTOR - EXPERT-2        JP)
 IFRUN BP>30.1
 ANDRUN BP-SLOW-FALL
 ANDRUN NEDIR
 BECAUSE WINDS FROM NE OR E
 AND SEASON IS WINTER
 BECAUSE NOT SUMMER
 AND CLOUDS INDICATE BAD WEATHER
THEN RAIN WITHIN 24 HRS
ANDTHEN WEATHER TURNING BAD
 IFRUN BP>30.1
 ANDRUN BP-RAPID-FALL
 ANDRUN NEDIR
 BECAUSE WINDS FROM NE OR E
 AND SEASON IS SUMMER
 BECAUSE NOT WINTER
 AND CLOUDS INDICATE BAD WEATHER
THEN RAIN WITHIN 12 - 24 HRS
ANDTHEN WEATHER TURNING BAD
-->


Screen 13
( WEATHER PREDICTOR - EXPERT-2        JP)
 IFRUN BP<29.8
 ANDRUN BP-RAPID-FALL
 ANDRUN NDIR
 BECAUSE WINDS FROM N, NE, OR E
THEN SEVERE STORM WARNING
ANDTHEN SEVERE NORTHEAST GALES
ANDTHEN WEATHER TURNING BAD
 IFRUN BP<29.8
 ANDRUN BP-RAPID-FALL
 ANDRUN SEDIR
 BECAUSE WINDS FROM E, SE, OR S
THEN SEVERE STORM WARNING
ANDTHEN RAIN OR SNOW IMMINENT
ANDTHEN WEATHER TURNING BAD
-->

Screen 14
( WEATHER PREDICTOR - EXPERT-2        JP)
 IFRUN BP<30.1
 ANDRUN BP-SLOW-FALL
 ANDRUN EDIR
 BECAUSE WINDS FROM SE, E, OR NE
 AND CLOUDS INDICATE BAD WEATHER
THEN RAIN FOR NEXT DAY OR TWO
ANDTHEN WEATHER TURNING BAD
-->

Screen 15
( WEATHER PREDICTOR - EXPERT-2        JP)
 IFRUN BP<30.1
 ANDRUN BP-RAPID-FALL
 ANDRUN EDIR
 BECAUSE WINDS FROM SE, E, OR NE
 AND CLOUDS INDICATE BAD WEATHER
THEN RAIN WITH HIGH CLOUDS
ANDTHEN CLEARING WITHIN 24 HRS
ANDTHEN COOLER TEMPERATURES
ANDTHEN WEATHER TURNING BAD
-->

Screen 16
( WEATHER PREDICTOR - EXPERT-2        JP)
 IFRUN BP>30.1
 ANDRUN BP-RAPID-RISE
 ANDRUN WDIR
 BECAUSE WINDS FROM SW, W, OR NW
THEN FAIR TODAY
ANDTHEN RAIN & WARMER NEXT 48 HRS
ANDTHEN WEATHER TURNING BAD
 IFRUN BP>30.1
 ANDRUN BP-SLOW-FALL
 ANDRUN WDIR
 BECAUSE WINDS FROM SW, W, OR NW
THEN WARMER
ANDTHEN RAIN WITHIN 24 - 36 HRS
ANDTHEN WEATHER TURNING BAD
-->

Screen 17
( WEATHER PREDICTOR - EXPERT-2        JP)
 IFRUN BP>30.1
 ANDRUN BP-RAPID-FALL
 BECAUSE RAPID FALL > .06" PER HOUR
 ANDRUN WDIR
 BECAUSE WINDS FROM SW, W, OR NW
 AND CLOUDS INDICATE BAD WEATHER
THEN WARMER
ANDTHEN RAIN WITHIN 18 - 24 HRS
ANDTHEN WEATHER TURNING BAD
 IFRUN BP>30.1
 ANDRUN BP-SLOW-FALL
 ANDRUN SDIR
 BECAUSE WINDS FROM SE OR S
THEN RAIN WITHIN 24 HRS
ANDTHEN WEATHER TURNING BAD
 IFRUN BP>30.1
 ANDRUN BP-RAPID-FALL
 ANDRUN SDIR
 BECAUSE WINDS FROM SE OR S
THENHYP WEATHER TURNING BAD
ANDTHEN WINDY, RAIN WITHIN 12 HRS
-->
Screen 18
( WEATHER PREDICTOR - EXPERT-2        JP)
( closing statement generator )
 IFNOT WEATHER OK
 ANDNOT WEATHER TURNING BAD
 ANDNOT WEATHER IMPROVING
THENHYP INSUFFICIENT DATA FOR A FORECAST

DONE
;S
```

End Listing

RSA: A Public Key Cryptography System, Part II

Welcome back for the second pass. If you are like I am, you have read or scanned between twenty and forty magazines in the month that has passed since you read the first part of this article. (It keeps getting harder to keep up with technology.) Therefore, we should briefly review what we presented in Part I.

Part I Revisited

We first learned that Diffie and Hellman introduced the Public Key System (PKS) in 1976[1] and expanded on it in 1979.[2] In 1978, Rivest, Shamir, and Adleman (RSA) proposed a method of PKS implementation and introduced signing messages with digital signatures.[3]

Then we ran through a brief introduction to RATFOR (RATional FORtran), the implementation language. RATFOR, which is similar in many ways to C and Pascal, adds several structured constructs to Fortran, and the output of the RATFOR precompiler is Fortran. Sample compilation of a RATFOR "program" to source was provided in Listings One to Three.

Next we reviewed modulo arithmetic. We saw that modulo arithmetic has some advantages and disadvantages. It limits the size of the numbers that are generated, but it requires considerable extra computation. As we will see, modulo arithmetic is an integral part of the RSA cryptosystem.

Next we looked at multiple-precision arithmetic. The multiple-precision arithmetic algorithms implemented in Part I form the core of the RSA system that we will present here. Knuth's algorithms[4] were stated and Listing Four showed the RATFOR source that implemented these algorithms. Four algorithms were presented: addition, subtraction, multiplication, and division. We saw that these algorithms were very computation intensive and that optimization efforts should probably be directed at them.

by C.E. Burton

C. E. Burton, 1720 S. Deframe Court, Denver, CO 80228.

Copyright © 1983 by C. E. Burton. All rights reserved. Permission is granted to use this software for personal, noncommercial use only.

Finally, the Russian Peasant exponentiation algorithm was presented. The basic algorithm found in Knuth's book had to be expanded to include modulo operations (a requirement of RSA). The Russian Peasant algorithm is most suitable for use with large exponents, which is why it was selected. Listing Five showed the RATFOR implementation of the algorithm.

In this part we will explore the generation and testing of large prime numbers, the generation of public/private keys, the encryption/decryption process, and the use of digital signatures. However, before we get to the meat of things, we must revisit RATFOR.

RATFOR Revisited

One of the advantages of RATFOR is that it allows character and string manipulation, a shortcoming of Fortran. Most of the library functions are associated with character manipulation. Since several of these library routines are used in the programs presented in this part, we will look at them briefly.

Also, RATFOR can have a global definition file, which I have called RATDEF.RAT. RATDEF contains definitions for most of the common ASCII characters, including control characters. For example:

```
define(BIGD,68) # "D"
define(LETD,100) # "d"
define(BACKSPACE,8)
    # <BACKSPACE> character
```

These are defined because Microsoft's Fortran converts all lower-case characters to upper case, i.e., "d" translates to "D." In addition, RATDEF contains some key word definitions. For example:

```
define(character,byte)
    # make CHARACTER type equivalent to BYTE
define(EOS,-2)
    # End of String character
define(EOF,-1) # End of File character
```

The library functions that are used in the RSA-PKS routines include:

- CLOSER - closes a file opened by OPENR
- EQUAL - compares two strings for equality
- ERROR - prints a string then exits to the operating system
- GETC - waits for and gets a character from the console
- GETCH - gets a character from a Logical Unit specified in OPENR
- OPENR - opens a character type file in the specified mode and sets aside a specified buffer size
- PUTCH - sends a character to the Logical Unit specified in OPENR
- PUTDEC - converts an integer to a character string of specified length and sends the string to the console
- REMARK - sends a string to the console without a carriage return; a period terminates the string
- RMRKLN - sends a string to the console terminated by a carriage return; a period terminates the string
- TYPE - determines if a character is a DIGIT or a LETTER

So much for our revisit to RATFOR. Now, let's talk a little bit about the RSA system and what makes it so advantageous.

Public Key Systems

PKS cryptography systems are different from most other cryptography systems in that a key is made public. Other cryptography systems require that the key be kept secret; otherwise, the system is compromised. Private key methods require that both the sending and the receiving sites have copies of the key so that they can encrypt/decrypt messages. The logistics of key distribution can be very awkward and costly. For communication networks, this process is almost useless.

PKS systems do not have these problems, since everyone has access to the public (encrypting) key; however, only the receiver knows the private (decrypting) key. This dual key approach allows anyone to use the public key to encrypt a message and send it to the holder of the private key, who is the only individual that can decrypt the message. For this system to work, knowledge of the public key cannot provide any knowledge about the private key, which would compromise the cryptography system.

The RSA system generates keys that have 200+ decimal digits by using prime numbers that have 100+ decimal digits. What makes the public key so secure is that efficient methods of factoring a very large number into its prime factors is beyond current mathematical feasibility, even using today's supercomputers. Estimates of factoring such numbers range from hundreds to billions of years, essentially an impossible task.

That is not to say that a mathematical

breakthrough cannot happen tomorrow and render the RSA system useless, similar to the recent breaking of the Knapsack System (a rival PKS). Knuth presents at least four methods of factoring numbers into their prime factors, including the sieve method that is used as a standard benchmark program. To factor a number, N, all we have to do is check all prime numbers less than or equal to the square root of N. If N is less than a million, there are only 168 primes to search. As numbers become larger, a prime number occurs about once every 2.3 times the number of decimal digits in the number. Thus, for 100-digit numbers, a prime number will be found once every 2300 numbers or so.

The problem then becomes: How do we find primes having 100+ decimal digits?

Large Prime Numbers

We are not really interested in factoring numbers into all their prime factors; we simply want to find a pair of prime numbers. Luckily, a theorem exists for this, although it does not say that a number is prime, only that it is not prime. The algorithm is based on Fermat's theorem, which states that if P is prime and X is not a multiple of P, then (X ** (P-1)) mod P = 1. Unfortunately, the theorem is not an if-and-only-if condition; i.e., it is not necessarily true that if the result is one, then P is prime. However, we *can* say that if the result is not one, then P is definitely not prime. (By the way, now we can see one of the reasons that we were interested in modifying the Russian Peasant algorithm to include modulo arithmetic.)

Knuth points out that some non-prime numbers can give prime testing routines great difficulty, such as (X ** 560) mod 561 = 1 for all X relatively prime to 561. To overcome this difficulty, Knuth provides a probabilistic prime test. If the algorithm is invoked K times, choosing X independently and randomly, and if the algorithm finds that the number is probably prime in all cases, then there is only one chance in (4 ** K) that the number is not prime. On the other hand, if during any pass the number is found to be non-prime by the algorithm, then the number is definitely not prime.

In Listing Six (page 40) you will find the prime test routine (PRIMTEST.RAT). It is not an implementation of Knuth's Algorithm P but is a simpler offshoot of the Fermat theorem, which we will call algorithm P′. Algorithm P′ proceeds as follows:

Prime Test Algorithm

Algorithm P′ (probabilistic primality test). Given an odd, positive integer, N, this algorithm attempts to decide whether or not N is prime.

P′1. [Check divisibility by 5] If the least significant digit of N is divisible by 5 (1sd(N) mod 5 = 0), then terminate the algorithm and say N is not prime.

P′2. [Generate exponent] Set M = N-1.

P′3. [Initialize] Set K = 1.

P′4. [Generate X] Let X be a random, positive integer in the range 1<X<N.

P′5. [Exponentiate] Set Y = (X ** M) mod N.

P′6. [Done?] If K = KTIMES and Y = 1, then terminate the algorithm and say N is probably prime.

P′7. [Not prime] If Y != 1, then terminate the algorithm and say N is not prime.

P′8. [Increase K] Set K = K+1 and return to Step P'4.

As with Knuth's Algorithm P, if this algorithm reports that N is not prime, then N is definitely not prime. If N is reported to be probably prime, a finite probability remains that N is not prime! I cannot say with any certainty what the probability is. I checked 561 with the above algorithm and, after fifty attempts to determine the primality of 561, the algorithm only reached five passes (K = 5) once; usually it only reached one or two passes. The algorithm could probably be made more efficient at Step P′1 by using some of the fast divisibility algorithms presented by H. T. Gordon.[5] It should be noted that KTIMES does not have to be 30, as is the case in Listing Six (see the *define* near the beginning of the listing), but reducing the value increases the probability that N will not be prime. The source code in Listing Six provides a test number count as the algorithm is executing (PUTDEC call) so that you can see how the operation is progressing.

Now that we can test for and generate large prime numbers, we are in a position to generate the public and private keys. So, let's look at the key generation procedure.

Key Generation

Before we dig into the generation of the keys, let's look at the RSA-PKS cryptography system in some detail to get a better feel for where we are heading. The RSA system is based on the use of large prime numbers, but in fact, we need only two large prime numbers, P and Q (at least 100 decimal digits each is preferable). The public (encryption) key is formed by the product of P and Q, i.e., N = P * Q. As you can imagine, factoring N is nearly impossible since N is a number composed of at least 200 decimal digits.

Next we now take the clear text and break it up into blocks; each block is formed by translating the characters into numbers and concatenating the digits to form a number, X, such that 0<X<N. To encrypt the message, we take each block, X, in turn, and form Y = (X ** 3) mod N. Each encrypted block, Y, is then transmitted to the intended receiver. The person receiving the message has a private (decryption) key, generated from P and Q. The private key, D, is formed by picking D such that D < N and (3 * D) mod ((P-1) * (Q-1)) = 1. We find that ((X ** 3) ** D) mod N = X; i.e., D is the cube root operator modulo N.

There is one further restriction on X. X must be greater than the cube root of N, otherwise we could easily decrypt the encrypted message using a standard cube root operation, since (X ** 3) mod N = X ** 3! What happens if one or more of the blocks does not satisfy this restriction? Well, we just pad the end of the block with random characters until the restriction is satisfied. Also, you might ask at this point, why can't I just take the cube root of Y and retrieve the clear text message, X? Well, think about what happened when you performed the modulo operation. The RSA-PKS operation is called a one-way, trap-door function, because it is easy to go one direction but not the other direction.

A couple of conditions on P and Q should be observed. First, (P-1) and (Q-1) cannot be divisible by 3; if (P-1) or (Q-1) is divisible by 3, then D may not decode the encrypted message because it does not produce a unique cube root. Second, (P-1) and (Q-1) should each have at least one large prime factor, otherwise the potential exists that N could be easily factored (see Knuth). Finally, P/Q should not be close to a simple fraction, or, again, N could be easily factored.

Now let's see how we can generate P and Q to get the keys. If you review the two previous paragraphs, you will agree that generating the keys is our most complex task (and the code will reflect this fact). Listing Seven (page 46) is the key generation routine (GENPKEYS.RAT). It uses every routine that we have discussed so far. Near the beginning of the program, some *defines* specify the KEYLENGTH and the BYTEMODULUS. These values are not set in concrete and can be reduced subject to some caution. BYTEMODULUS can be any value between 2 and 128, at the potential cost of increasing computation time in the prime testing routine. KEYLENGTH can be reduced at the risk of increasing the key's vulnerability. The length of 90 is equivalent to a 630-bit key length ((2 ** 7) ** 90) or a 190-decimal-digit key.

Our first task is to seed the random number generator. We must do this because the Microsoft random number routine always generates the same sequence of numbers if not seeded, thus giving everyone the same set of keys. We can enter any number from 1 to 9999999. The length of the seed is probably the main weakness of this RSA implementation. Given enough resources and the

Microsoft random number generation routine algorithm, I theoretically could go to any supercomputer or use a room full of microcomputers to generate all 10 million key sets. Then when you publish your public key, I could look it up, get the corresponding private key, and decrypt any of your messages. You could make my task more difficult by changing the value of KEYLENGTH and the associated values for the length of P and Q.

The program randomly selects the length of P and Q while assuring that their ratio is not a simple fraction. Then both P and Q are generated by the same routine, GENPRM. The algorithm used by GENPRM follows.

Generate Prime Algorithm

Algorithm GENPRM (generate a prime number - after Knuth). This algorithm generates a prime number of length Len(PRIME).

G1. [Get length of PO and K] Set Len(PO) = 2 * Len(PRIME) / 3 and Len(K) = Len(PRIME) - Len(PO). Here, PO will end up being a prime number used to generate PRIME, and K will be a multiplying factor used to generate PRIME.

G2. [Generate random PO and K] Generate a random value of PO of length Len(PO). Generate a random value of K of length Len(K).

G3. [Make PO odd and K even] If PO is even, set PO = PO + 1. If K is odd, set K = K - 1. (PO must be odd to be a prime.)

G4. [PO prime?] Test primality of PO. If PO is probably prime, go to Step G6. (Approximately Ln(PO)/2 numbers will be tested before finding a prime.)

G5. [PO not prime] Set PO = PO + 2 and repeat Step G4.

G6. [Adjust K] Find K such that K is even and K mod 3 = PO mod 3.

G7. [Get trial PRIME] Set PRIME = K * PO + 1.

G8. [PRIME prime?] Test primality of PRIME. If PRIME is probably prime, go to Step G10. (Approximately Ln(K * PO)/6 numbers will be tested before finding a prime.)

G9. [PRIME not prime] Set PRIME = PRIME + (6 * PO) and repeat Step G8. (Keeping K mod 3 = PO mod 3, K even and PRIME = K * PO + 1, implies K = K + 6 at this iteration. Substituting the new K into the equation for PRIME produces the above PRIME adjustment.)

G10. [Check PRIME + 1] Find greatest common divisor of (PRIME + 1). If the greatest prime factor of (PRIME + 1) is less than 100,000, then go to Step G9. Otherwise, terminate the algorithm and return PRIME.

With P and Q determined, N and D can be calculated. The public key, N, is found by N = P * Q and is of length Len(P) + Len(Q) = KEYLENGTH. The private key, D, is found by picking D such that D < N and (3 * D) mod (P-1) * (Q-1) = 1. This statement is equivalent to D = 2 * ((P-1) * (Q-1) + 1)/3. We now have our keys. After all of this trouble, we certainly want to save them so that they can be used in the encryption/decryption process.

Finally, we are ready to look at the encryption/decryption process. This process is the topic of the next section.

```
Encryption Key - Len(N) & N
 20
 26 26 45 90 98 127 6 31  90 33 86 86 13 99 35 121 110 14 44 77

Decryption Key - Len(D) & D
 20
 17 60 30 60 65 127 46 49 0 90 98 46 40 95 60 13 54 79 10 83

First Prime - Len(P) & P
  7
 38 117 16 61 54 24 123

Second Prime - Len(Q) & Q
 13
 86 25 25 67 16 80 45 20 85 30 97 67 87
```

Figure 1A.

Keys and Primes generated by GENPKEYS. KEYLENGTH = 20 and KTIMES = 5.

```
This is an example of th RSA encryption/decryption algorithm. This example
uses a KEYLENGTH of 20 (140 binary digits), a P length of 49 binary digits,
and a Q length of 91 binary digits. The random number seed was 1111111.
The number of prime tests was reduced from 30 to 5. It took approximately
00:52:38 to generate the keys. It took about 00:08:02 to encrypt this file and
03:18:46 to decrypt the encrypted file.

For a RSA with a KEYLENGTH of 90 (630 binary digits), it takes approximately
11 days to generate the keys. The computer used to perform the algorithm was a
Z80-based machine running at 4Mhz under MP/M 2.0 (single user).
```

Figure 1B.

Clear (Plain) Text to be encrypted.

Encryption/Decryption

This section will be fairly short because we have already taken a look at the encryption/decryption process in the beginning of the last section. Listing Eight (page 56) depicts the encryption/decryption process. Again, the definitions of BYTEMODULUS and KEYLENGTH appear near the start of EDCRYPT.RAT. Note that they must be the same values used in the key generation process!

We start with a clear text message X = {x1, x2, . . . , xn} where 0 < xk < N. If xn, the final block, is less than the cube root of N, terminate the block with an End of File character and pad xn by concatenating random characters to the end of the block. I use ASCII control characters except ^Z, which is the CP/M End of File character. To encrypt each block, xk, we look at each character of xk and strip off the most significant bit; this ensures that 0 < char(xk) < BYTEMODULUS. Next we form yk = (xk ** 3) mod N. Finally, we scan each character of yk and set the most significant bit if it is an ASCII control character. This action prevents a ^Z or some other character from giving the RATFOR library character I/O routines a problem. The encrypted blocks Y = {y1, y2, . . . , yn} are sent to the intended receiver. Note that each encrypted block, yk, is of length Len(N).

As the authorized receiver, we can now apply our private key to decode the received message. We reblock Y into blocks of length Len(N). We take each block, yk, in turn. Each character (byte)

is scanned and the most significant bit is stripped, returning the character to its original encrypted form. We now apply the inversion operation, xk = (yk ** D) mod N. The final block, xn, is scanned from the last character to the first character, looking for the End of File character to find the end of the original text. The blocks X = {x1, x2, . . . , xn} are assembled into the decrypted, clear text. EDCRYPT.RAT performs these basic operations, as well as selecting the files to encrypt/decrypt and the necessary file I/O.

Figure 1 (below) shows an example of generated keys, clear text, encrypted text, and decrypted text. The values of KEYLENGTH and KTIMES have been reduced to permit a reasonable amount of execution time: KEYLENGTH was set to 20 instead of 90 and KTIMES was set to 5 instead of 30. It took almost an hour to find the keys. The encryption of CLEARO.TXT into ENCRYPTO.TXT took about 8 minutes and the decryption of ENCRYPTO.TXT to DECRYPTO.TXT took nearly 3 hours and 19 minutes. When I used the original values of KEYLENGTH and KTIMES, I found that it took almost 11 days to find the keys.

To test how much an arithmetic coprocessor might increase performance, I used an AMD 9511 and the Redding Group APU Fortran library. Unfortunately, the library did not include a random number generator and the Microsoft module would not work with the APU library. Therefore, I had to substitute the random number routine found in Listing Nine (page 59). With these changes and using the reduced example above, I found that the routines executed in about half the time (key generation in 1:20, encryption in 0:04, and decryption in 1:33). The key generation took longer because the random number generator was not the same and different keys were produced. Encryption/decryption, however, used the same keys produced by the non-APU version to keep comparisons as close as possible. As I have pointed out before, optimizing the multiple-precision arith-

```
81 7F 6B 85 22 85 21 66  8E 40 99 3A 5A 35 8D 40   .^k.".!f.@.:Z5.@
65 9D 9D 6B 82 3C 94 74  25 8E 72 47 68 2B 50 30   e..k.<.t%.rGh+P0
68 62 6A 4D 2D 30 7A 87  8A 9C 59 28 4A 6F 5C 2C   hbjM-0z...Y(Jo\,
8E 99 20 8E 4D 45 63 28  83 7E 32 6A 98 43 2A 8D   .. .MEc(.~2j.C*.
45 70 5C 4A 61 3A 4F 63  92 6E 5F 90 44 9A 69 2E   Ep\Ja:Oc.n_.D.i.
89 82 87 9D 6F 82 6E 7F  52 61 29 4A 95 75 9A 37   ....o.n~Ra)J.u.7
49 8D 2B 70 82 85 29 98  5E 67 9E 40 25 62 9B 37   I.+p..).^g.@%b.7
9B 60 91 44 4A 73 7C 6C  83 90 4E 94 58 36 78 70   .`.DJs|l..N.X6xp
46 56 95 75 2B 82 52 7A  34 9C 46 8B 88 7A 45 3B   FV.u+.Rz4.F..zE;
59 98 38 29 8B 3C 9C 71  44 64 2A 48 7E 26 6C 7D   Y.8).<.qDd*H~&l}
8D 6B 72 3D 52 55 4B 28  4A 3A 48 6B 49 39 81 9B   .kr=RUK(J:HkI9..
43 57 42 67 8C 4C 9D 5F  58 43 2E 39 8E 34 48 36   CWBg.L._XC.9.4H6
3F 89 2F 49 97 62 86 67  93 85 42 8C 55 38 58 49   ?./I.b.g..B.U8XI
9E 46 58 78 21 6C 63 42  38 25 9C 59 8F 6C 25 7B   .FXx!lcB8%.Y.l%{
77 4E 82 33 51 65 86 98  32 88 5A 2E 83 42 6C 33   wN.3Qe..2.Z..Bl3
94 5E 56 6B 8D 33 83 51  8A 25 87 4F 99 64 99 83   .^Vk.3.Q.%.O.d..
7F 43 5A 58 8F 23 9C 8A  75 68 70 73 7F 3E 9F 80   ~CZX.#..uhps~>..
5E 75 8F 5F 3E 7B 63 43  8B 87 6F 44 83 24 5E 48   ^u._>{cC..oD.$^H
89 7F 89 81 71 5C 84 37  61 5F 63 41 8D 34 62 8C   .~..q\.7a_cA.4b.
3E 3C 83 42 58 29 64 72  56 46 60 63 7E 42 9D 79   ><.BX)drVF`c~B.y
8B 22 93 20 2A 71 37 6D  70 9E 63 99 2F 2D 61 3A   ."." *q7mp.c./-a:
57 4A 85 25 80 64 87 52  39 7E 4D 38 7B 36 89 2F   WJ.%.d.R9~M8{6./
59 6F 41 6A 76 7B 91 57  8B 8B 52 64 31 92 65 4F   YoAjv{.W..Rd1.eO
50 35 90 90 91 6B 57 7C  51 87 9F 9E 97 89 40 77   P5...kW|Q.....@w
67 74 4E 2A 34 31 65 7D  3D 97 63 9C 67 61 99 52   gtN*41e}=.c.ga.R
92 8A 99 43 97 6A 4E 44  57 91 26 7F 8A 78 2A 6C   ...C.jNDW.&~.x*l
67 54 3E 6B 84 75 94 88  35 71 56 90 55 51 41 30   gT>k.u..5qV.UQAO
61 2B 8F 4D 3D 43 6C 81  89 2F 22 69 9B 5E 6A 8B   a+.M=Cl../"i.^j.
61 77 95 36 75 3E 6A 67  7F 21 5C 4C 84 63 20 93   aw.6u>jg~!\L.c .
77 7E 6D 86 21 7A 8B 59  64 3C 5E 4F 87 7F 39 29   w~m.!z.Yd<^O.~9)
8C 8C 5A 45 8C 36 9E 4E  29 61 73 63 2D 7F 7C 33   ..ZE.6.N)asc-~|3
6D 86 25 94 86 62 7F 8E  3F 44 7D 35 5B 7D 66 27   m.%..b~.?D}5[}f'
7E 7F 99 9F 71 5C 30 34  88 51 25 80 78 43 90 8D   ~~..q\04.Q%.xC..
49 8A 9D 99 48 95 94 8C  63 38 2D 2E 96 9E 26 70   I...H...c8-...&p
2B 46 4E 50 3D 7D 81 59  66 52 87 45 3B 40 26 93   +FNP=}.YfR.E;@&.
80 82 4C 8A 3E 37 4A 81  88 6D 24 2D 25 8F 65 85   ..L.>7J..m$-%.e.
20 67 45 90 97 66 7D 67  39 6F 20 23 29 46 72 7F    gE..f}g9o #)Fr~
97 85 70 47 63 9B 3B 41  8D 27 44 87 56 50 2A 41   ..pGc.;A.'D.VP*A
2B 45 45 6E 5F 74 79 44  85 2A 23 8B 91 2E 6C 89   +EEn_tyD.*#...l.
8B 4E 8F 76 8E 24 94 9F  42 85 6D 78 9A 63 48 22   .N.v.$..B.mx.cH"
92 2B 6F 5C 71 34 43 59  6A 56 96 30 47 9F 25 30   .+o\q4CYjV.0G.%0
9D 35 85 8A 87 3C 88 68  35 62 81 93 84 5D 97 61   .5...<.h5b...].a
51 7C 87 5D 67 93 66 7C  1A                        Q|.]g.f|.
```

Figure 1C.

Encrypted Text using Encryption Key in Figure 1A and Clear Text in Figure 1B.

metic and Russian Peasant routines could also have a great impact on the performance. However, I will leave that as an exercise for the reader.

One final topic remains. How can we sign our messages so that the receiver knows without a doubt that we sent the message? Digital signatures provide the answer.

Digital Signatures

Now, let's consider a little "Spy vs. Spy," a la *Mad Magazine*. White 1 wants to send secure messages to White 2. However, Black is always capable of intercepting the transmitted messages and substituting his own messages. The ability of Black to intercept and substitute causes some significant problems for White, e.g., risk of dirty tricks and crosses over the eyes. The problem is: How can White 2 be sure that a received message from White 1 is the real thing and not a setup?

A plain text signature is no good, since Black could sign the message with White 1's name before encrypting the message with White 2's public key. How can White use the RSA-PKS system to confirm that a message was sent from a friend? Well, assume that there are public keys that are published in something like the phone book. Anyone is able to take his plain text message and pass it through the encryption algorithm using another's published, public key.

Let's go back to the section on modulo arithmetic and look at the commutative property of exponentiation. Eureka! We find that:

```
X = ((X ** 3) ** D) mod N
  = (((X ** 3) mod N) ** D) mod N
  = (((X ** D) mod N) ** 3) mod N
```

That is, the order of encryption and decryption algorithm application does not matter. Thus, White 1 can take his clear text message, X, and encrypt it with his private key, Dm, then encrypt the result further with White 2's public key, Ni, and finally send the resultant message, Y, to White 2.

```
Ysig = (X ** Dm) mod Nm
                        <- Signature Step
Y = (Ysig ** 3) mod Ni
                        <- Encryption Step
```

Black cannot read the message! To read the message, White 2 reverses the

```
54 68 69 73 20 69 73 20  61 6E 20 65 78 61 6D 70   This is an examp
6C 65 20 6F 66 20 74 68  65 20 52 53 41 20 65 6E   le of the RSA en
63 72 79 70 74 69 6F 6E  2F 64 65 63 72 79 70 74   cryption/decrypt
69 6F 6E 0D 0A 61 6C 67  6F 72 69 74 68 6D 2E 20   ion..algorithm.
20 54 68 69 73 20 65 78  61 6D 70 6C 65 20 75 73    This example us
65 73 20 61 20 4B 45 59  4C 45 4E 47 54 48 20 6F   es a KEYLENGTH o
66 20 32 30 20 28 31 34  30 0D 0A 62 69 6E 61 72   f 20 (140..binar
79 20 64 69 67 69 74 73  29 2C 20 61 20 50 20 6C   y digits), a P l
65 6E 67 74 68 20 6F 66  20 34 39 20 62 69 6E 61   ength of 49 bina
72 79 20 64 69 67 69 74  73 2C 20 61 6E 64 0D 0A   ry digits, and..
61 20 51 20 6C 65 6E 67  74 68 20 6F 66 20 39 31   a Q length of 91
20 62 69 6E 61 72 79 20  64 69 67 69 74 73 2E 20    binary digits.
20 54 68 65 20 72 61 6E  64 6F 6D 20 6E 75 6D 62   The random numb
65 72 0D 0A 73 65 65 64  20 77 61 73 20 31 31 31   er..seed was 111
31 31 31 31 2E 20 20 54  68 65 20 6E 75 6D 62 65   1111.  The numbe
72 20 6F 66 20 70 72 69  6D 65 20 74 65 73 74 73   r of prime tests
20 77 61 73 0D 0A 72 65  64 75 63 65 64 20 66 72    was..reduced fr
6F 6D 20 33 30 20 74 6F  20 35 2E 20 20 49 74 20   om 30 to 5.  It
74 6F 6F 6B 20 61 70 70  72 6F 78 69 6D 61 74 65   took approximate
6C 79 20 30 30 3A 35 32  3A 33 38 0D 0A 74 6F 20   ly 00:52:38..to
67 65 6E 65 72 61 74 65  20 74 68 65 20 6B 65 79   generate the key
73 2E 20 20 49 74 20 74  6F 6F 6B 20 61 62 6F 75   s.  It took abou
74 20 30 30 3A 30 38 3A  30 32 20 74 6F 0D 0A 65   t 00:08:02 to..e
6E 63 72 79 70 74 20 74  68 69 73 20 66 69 6C 65   ncrypt this file
20 61 6E 64 20 30 33 3A  31 38 3A 34 36 20 74 6F    and 03:18:46 to
20 64 65 63 72 79 70 74  20 74 68 65 20 65 6E 63    decrypt the enc
72 79 70 74 65 64 0D 0A  66 69 6C 65 2E 0D 0A 0D   rypted..file....
0A 46 6F 72 20 61 20 52  53 41 20 77 69 74 68 20   .For a RSA with
61 20 4B 45 59 4C 45 4E  47 54 48 20 6F 66 20 39   a KEYLENGTH of 9
30 20 28 36 33 30 20 62  69 6E 61 72 79 20 64 69   0 (630 binary di
67 69 74 73 29 2C 20 69  74 0D 0A 74 61 6B 65 73   gits), it..takes
20 61 70 70 72 6F 78 69  6D 61 74 65 6C 79 20 31    approximately 1
31 20 64 61 79 73 20 74  6F 20 67 65 6E 65 72 61   1 days to genera
74 65 20 74 68 65 20 6B  65 79 73 2E 20 20 54 68   te the keys.  Th
65 0D 0A 63 6F 6D 70 75  74 65 72 20 75 73 65 64   e..computer used
20 74 6F 20 70 65 72 66  6F 72 6D 20 74 68 65 20    to perform the
61 6C 67 6F 72 69 74 68  6D 20 77 61 73 20 61 20   algorithm was a
5A 38 30 2D 62 61 73 65  64 0D 0A 6D 61 63 68 69   Z80-based..machi
6E 65 20 72 75 6E 6E 69  6E 67 20 61 74 20 34 4D   ne running at 4M
68 7A 20 75 6E 64 65 72  20 4D 50 2F 4D 20 32 2E   hz under MP/M 2.
30 20 28 73 69 6E 67 6C  65 20 75 73 65 72 29 2E   0 (single user).
0D 0A 1A                                           ...
```

Figure 1D.

Decrypted Text using Decryption Key in Figure 1A and Encrypted Text in Figure 1C.

process. The message, Y, is passed through White 2's private algorithm, Di, and a yet garbled message is found. White 2 suspects that White 1 sent the message, so White 2 passes the resultant message through White 1's encryption algorithm, Nm, and, lo and behold, White 1's plain text message, X, appears.

```
Ysig = (Y ** Di) mod Ni
                    <- Decryption Step
X = (Ysig ** 3) mod Nm
          <- Signature Verification Step
```

White 2 is now sure that the message came from White 1 because no one else could know Dm! Black could not make a substitution because he cannot know White 1's private key, unless White 1 has sloppy security procedures.

To review the process, the encryption of a message using digital signatures consists of:

```
Y = (((X ** Dm) mod Nm) ** 3) mod Ni
```

The decryption process consists of:

```
X = (((Y ** Di) mod Ni) ** 3) mod Nm
```

That is all there is to it. For further information on the use of digital signatures, you should read the articles in IEEE *Computer Magazine*, February 1983.[6-9]

Summary

We have covered a lot of material in these two parts. In the first part we looked at RATFOR, modulo arithmetic, multiple-precision arithmetic, and the Russian Peasant algorithm. We saw that these routines form the core of the RSA-PKS cryptography system.

In this second part we revisited RATFOR to look at the RATDEF file and some of the RATFOR library routines. The RATDEF file contains some basic keyword and character definitions. The library routines add character I/O to the basic Fortran library.

Next, we looked at Public Key Systems and found what made them advantageous. We do not have to worry about key distribution, because the public key can be made accessible to anyone without worrying about compromising the encryption/decryption process. The public key does not provide any information about the private key.

Third, we looked at the generation of large prime numbers. We used Fermat's theorem to test odd, positive numbers for primality. The test was a probabilistic test that has a finite probability that a reported prime number is not prime, even though the probability is very small. However, if a number is said to be not prime, then it is definitely not prime.

Fourth, we looked at the generation of the public and private keys. We used all of the previously developed routines to generate the prime factors of the keys, P and Q. These prime numbers were then used to generate the public and private keys, N and D, respectively.

Next, we used the keys to encrypt and decrypt plain text messages. We saw how to block out the messages to encode or decode them. We found that we must pad the last block if it is too short to keep someone from easily decrypting it. Also, we found that using an APU can halve the time required to perform the encryption/decryption process.

Finally, we looked at digital signatures. We saw that another person could sign a message digitally by encrypting the message initially with their private key and then encrypting the result with the intended receiver's public key. The receiver reverses the process to confirm that the message was sent by the signee.

I hope you have enjoyed this article and found it understandable and educational. I owe Donald Knuth most of the credit for this article, since his book was the major contributor to my understanding of the RSA-PKS cryptography system. Of course, this article would not have been possible without the basic papers of Diffie and Hellman and of Rivest, Shamir, and Adleman. As I stated previously, I am sure there are better ways to implement the algorithms and even better and more efficient algorithms that could be used to improve the performance.

DDJ

(Continued on page 93)

RSA (Text begins on page 30)

Listing Six

Copyright © 1983, Charles E. Burton, Denver, Colorado.
All rights reserved. Permission granted to use this software for personal, noncommercial purposes only.

```
# PROGRAM NAME:  PRIMTEST.RAT
# PURPOSE:  This routine is a probabilistic test for the primality of a number.
#           If the test is true, there is a probability of (1/4)**K that the
#           number is not prime.  The test is based on Fermat's Theorem.
#           (re. D.E. Knuth, The Art of Computer Programming, V. 2
#           (Semi-Numerical Algorithms), 2nd Ed.,(Addison-Wesley, Reading, MA),
#           pp. 374-380.)
#
# LANGUAGE:  RATFOR
# AUTHOR:  CEB
# USAGE:  LOGCL=PRMTST(PRIMEQ,LENP,WORK,LENW,MODULO)
#               <PRMTST -- Logical function, indicates that the number in
#                          question (PRIMEQ) is probably prime with probability
#                          1-(1/4)**K.  PRMTST = .TRUE. if probably prime,
#                          and .FALSE. if it is definately not prime.
#               >PRIMEQ -- Byte array, contains the number (byte modulus MODULO)
#                          to be tested.  The MSDigit(s) are in PRIMEQ(1) and
#                          the LSDigit(s) are in PRIMEQ(LENP).
#                          CAUTION:  PRIMEQ > 2
#               >LENP -- Integer variable, defines length of PRIMEQ array.
#               <WORK -- Byte array, a working array needed to do test.
#               >LENW -- Integer variable, defines length of WORK array.
#                        CAUTION:  LENW = 7 * LENP + 3
#               >MODULO -- Integer variable, defines the arithmetic modulus that
```

(Continued on next page)

RSA (Listing Continued, text begins on page 30)

Listing Six

```
#                          is to be used.  MODULO has a byte-wide effect and
#                          should be between 2 and 128 (e.g. 100 for Decimal
#                          Numbers and 128 for ASCII Characters).
#
# ARRAYS USED: PRIMEQ(*)
# EXTERNALS:  RPEXP,MPSUBT
# UPDATE HISTORY:  INITIAL RELEASE -- 01/25/83 CEB
#
define(KTIMES,30) # number of times, K, to test PRIMEQ

logical function prmtst(primeq,lenp,work,lenw,modulo)

     character back3(4)
     byte primeq(1),work(1),one,notprm
     logical xflag
     integer lenp,lenw,modulo

     data back3/BACKSPACE,BACKSPACE,BACKSPACE,EOS/

     data one/1/
     data rndnxt/.5/ # make RAN(*) generate the next random number
                     #  (any positive number will do)
                     #  *** could seed it with a prior call to RAN(-.xxxxxx)
                     #      where <xxxxxx> is entered by the user ***

     prmtst=.false. # initialize to indicate not prime
     #######
     #
     # The following test assumes PRIMEQ != 5 (i.e. the prime number 5 is not
     #  sought).  If assumption is invalid them remove this test.

     lsdig=primeq(lenp) # get the LSDigit of PRIMEQ
     if ((modulo > 5) & (mod(lsdig,5) == 0)) # PRIMEQ divisible by 5?
          return # no need to check further so Exit !!!

     #
     #######
     idxprm=1 # initialize pointer to 1st digit of PRIMEQ
     while (primeq(idxprm) == 0 & idxprm <= lenp) # digit is zero
          idxprm=idxprm+1 # move index to next position
     lenp0=lenp-idxprm+1 # get length of right justified prime
     if ((lenw >= 7*lenp0+3) & (lenp0 >= lenp) &
         ((lenp0 != 1) | (primeq(1) > 2))) # work array has sufficient length
                                           #  and PRIMEQ > 2?
          {
          idxexp=lenp0+1 # get index to exponent
          call mpsubt(primeq(idxprm),lenp0,one,1,work(idxexp),lenp0,modulo)
                                        # generate (PRIMEQ - 1), i.e. exponent
          idxrlt=idxexp+lenp0 # get index to RESULT: X ** (PRIMEQ-1) mod PRIMEQ
          idxscr=idxrlt+lenp0 # get index to scratch work area for calculations
          fmodul=float(modulo) # float MODULO
          notprm=NO # initialize Not Prime flag
          call remark('     Test Number:  .') # indicate test running
          do i=1,KTIMES # test for primality
               {
               call putdec(i,3) # show test number
               call remark(back3) # backspace 3
               idxx=1 # initialize index to X
               idigit=primeq(idxprm) # get 1st digit of PRIMEQ
               if (idigit == 1) # first digit a 1?
                    {
                    xflag=.true. # indicate 1st digit of X is zero
```

```
                ix=0 # force X < PRIMEQ
                }
           else # first digit > 1
                {
           xflag=.false. # indicate 1st digit of X is non-zero
           repeat # get first digit of X
                {
                ix=mod(int(fmodul*ran(rndnxt)),idigit)
                                                  # force X < PRIMEQ
                }
           until ((lenp0 != 1 & ix != 0) | (lenp0 == 1 & ix > 1))
                              # 1st digit of X is valid
           }
      work(idxx)=ix # initialize 1st digit of X
      idxx=idxx+1 # move index to next position
      while (idxx <= lenp0) # generate a random integer
                            #  1 < X < PRIMEQ
           {
           if (xflag) # second digit of X needs special handling?
                {
                xflag=.false. # indicate 2nd digit of X is non-zero
                repeat # get second digit of X
                     {
                     ix=mod(int(fmodul*ran(rndnxt)),modulo)
                                                  # force X >= 2
                     }
                until ((lenp0 != 2 & ix != 0) | (lenp0 == 2 & ix > 1))
                               # 2nd digit of X is valid
                }
           else # PRIMEQ < X < 1

                ix=mod(int(fmodul*ran(rndnxt)),modulo) # get next
                                                       #  digit
           work(idxx)=ix # load next digit of X
           idxx=idxx+1 # move index to next position
           }
      call rpexp(work,lenp0,work(idxexp),lenp0,work(idxrlt),lenp0,
                 primeq(idxprm),lenp0,work(idxscr),4*lenp0+3,modulo)
                     # generate RESULT = X ** (PRIMEQ -1) mod PRIMEQ
      #
      #  RESULT = (X ** (PRIMEQ-1) mod PRIMEQ) != 1 --> not prime
      #
      idxwrk=idxscr-2 # get index to Digit above LSDigit of RESULT
      if (work(idxwrk+1) != 1) # LSDigit of RESULT --> not prime?
           notprm=YES # indicate Not Prime
      else # check rest of Digits
           {
           while (idxwrk >= idxrlt) # check the rest of RESULT's
                                    #  Digits
                {
                if (work(idxwrk) != 0) # not prime?
                     {
                     notprm=YES # indicate Not Prime
                     break # leave WHILE loop
                     }
                idxwrk=idxwrk-1 # move index to next position
                }
           }
           if (notprm == YES) # not prime?
                break # leave DO loop
           }
      if (notprm == NO) # high probability PRIMEQ is a prime?
           prmtst=.true. # indicate probably prime
      }
 else # cannot test PRIMEQ
      {
```

(Continued on next page)

Listing Six

```
            call rmrkln('.')
            call error('Length of [WORK] too small or [PRIMEQ] <= 2 !!!.')
                                        # print message and exit
            }
        return

end
```

End Listing Six

Listing Seven

Copyright © 1983, Charles E. Burton, Denver, Colorado.
All rights reserved. Permission granted to use this software for personal, noncommercial purposes only.

```
# PROGRAM NAME:  GENPKEYS.RAT
# PURPOSE:  This routine generates the public keys for the RSA
#           (Rivest-Shamir-Adleman) Public Key Encryption/Decryption System.
#           It uses the method described by D.E. Knuth, The Art of Computer
#           Programming, V. 2 (Semi-Numerical Algorithms), 2nd Ed., (Addison-
#           Wesley, Reading, MA), pp. 386-389.
#
# LANGUAGE:  RATFOR
# AUTHOR:  CEB
# USAGE:  ******** (main program)
#
#
#
# ARRAYS USED:  P(KEYLENGTH),Q(KEYLENGTH),N(KEYLENGTH),D(KEYLENGTH),
#               WORK(WORKLENGTH)
# EXTERNALS:  MPADD,MPSUBT,MPMULT,MPDIV,RPEXP,PRMTST
# UPDATE HISTORY:  INITIAL RELEASE -- 01/28/83 CEB
#

define(KEYLENGTH,90) # length of key for characters (assume 80 char. lines)
define(WORKLENGTH,810) # length of WORK buffer for characters
                       #  (9 * KEYLENGTH)
define(BYTEMODULUS,128) # byte modulus for characters (1 character/byte)

define(LUNOUT,6) # output logical unit

program pubkey

     character type
     byte p(KEYLENGTH),q(KEYLENGTH),n(KEYLENGTH),d(KEYLENGTH),work(WORKLENGTH),
          one,two,three
     double precision seed # allow entry of up to 8 characters

     equivalence (seed,work(1))

     data one,two,three/1,2,3/
     data rndnxt/0.5/ # positive number needed to generate next random number
     data modulo/BYTEMODULUS/ # get byte modulus for arithmetic


     write(CONSOLE,100)
100  format(1x,'Enter a seed for the random number generator (XXXXXXX):  ')
     read(CONSOLE,200) seed
200  format(a7)
     rnd1st=0.0 # initialize random number seed
```

```
do i=1,7 # generate random number generator seed
     {
     if (type(work(i)) == DIGIT) # entry is a digit?
          {
          inum=work(i)-DIGO # get value of seed digit
          rnd1st=10.0*rnd1st+float(inum) # add digit to seed
          }
     else # non-digit found
          break # exit DO loop
     }
rnd1st=-rnd1st/1.0e7 # -1.0 < seed <= 0
call ran(rnd1st) # seed random number generator
len=KEYLENGTH # get key length desired (i.e. length of P * Q)
lendel=4+int(3.0*ran(rndnxt)) # get a delta length between 4 and 7
                              #  (i.e. P and Q length diff. of 8 and 12)
lenp=len/2-lendel # get desired length of prime factor P
lenq=len-lenp # get desired length of prime factor Q
call rmrkln('.')
call rmrkln('Generating P.')
call genprm(p,lenp,work,modulo) # generate prime factor P
call rmrkln('.'); call rmrkln('.')
call rmrkln('Generating Q.')
call genprm(q,lenq,work,modulo) # generate prime factor Q
#
# generate the Public Key encryptor:
#   N = P * Q
#
call rmrkln('.'); call rmrkln('.')
call rmrkln('Generating N.')
call mpmult(p,lenp,q,lenq,n,len,modulo)
idx1=1 # get index to 1st digit of N
while (n(idx1) == 0) # find 1st non-zero digit of N
     idx1=idx1+1 # advance to next position
if (idx1 > 1) # need to shift N to make 1st digit non-zero (since modulus
              #  of exponential operation cannot have a leading zero)?
     {
     len=len-(idx1-1) # adjust length of N
     do idx2=1,len # left justify N, stripping zeroes
          {
          idx3=idx1+idx2-1 # get index to digit to move
          n(idx2)=n(idx3) # left justify N
          }
     }
#
# generate Public Key decryptor:
#   D = 2 * ((P - 1) * (Q - 1) + 1) / 3 = 2 * (N - P - Q) / 3 + 1
#
call rmrkln('.')
call rmrkln('Generating D.')
len1=max0(lenp,lenq)+1 # get length of (P + Q)
call mpadd(p,lenp,q,lenq,work,len1,modulo) # generate (P + Q)
idx1=len1+1 # get index to result of N - (P + Q)
call mpsubt(n,len,work,len1,work(idx1),len,modulo) # generate N - (P + Q)
idx2=idx1+len # get index to start of result of 2 * (N - P - Q)
len2=len+1 # get length of result of 2 * (N - P - Q)
     call mpmult(work(idx1),len,two,1,work(idx2),len2,modulo)
                                          # generate 2 * (N - P - Q)
     work(1)=0 # insure Numerator starts with 0 for division normalization
     do idx=1,len2 # copy Numerator <2 * (N - P - Q)>
          {
          work(idx+1)=work(idx2)
          idx2=idx2+1 # advance index to next position
          }
```

(Continued on next page)

Listing Seven

```
      len1=len2+1 # account for extra digit in numerator
      idx2=len1+1 # get index to quotient of <2 * (N - P - Q) / 3>
      len2=len1-1 # get length of quotient
      idx3=idx2+len2 # get index to remainder of <2 * (N - P - Q) / 3>
      call mpdiv(work,len1,three,1,work(idx2),len2,work(idx3),1,modulo)
                              # generate 2 * (N - P - Q) / 3 (i.e. Quotient)
      len3=len2+1 # get length of result of 2 * (N - P - Q) / 3 + 1
      call mpadd(work(idx2),len2,one,1,work(idx3),len3,modulo)
                              # generate 2 * (N - P - Q) / 3 + 1
      do idx=1,len # copy 2 * (N - P - Q) / 3 + 1 to D from LSDigit to MSDigit
                   #  Note: D < N, so Length(D) = Length(N)
                   #        LEN3 = LEN + 3
           {
           idx1=len-idx+1 # get index to D
           idx2=idx3+len3-idx # get index to 2 * (N - P - Q) / 3 + 1
           d(idx1)=work(idx2) # move 2 * (N - P - Q) / 3 + 1 to D
           }
      # send Keys to CONSOLE
      call rmrkln('.')
      call rmrkln('Keys and Key Factors.')
      write(CONSOLE,300) len # N
300   format(1x,20(i3,1x))
      write(CONSOLE,300) (n(idx), idx=1,len)
      write(CONSOLE,300) len # D
      write(CONSOLE,300) (d(idx), idx=1,len)
      write(CONSOLE,300) lenp # P
      write(CONSOLE,300) (p(idx), idx=1,lenp)
      write(CONSOLE,300) lenq # Q
      write(CONSOLE,300) (q(idx), idx=1,lenq)
      # send Keys to LUNOUT file
      call open(LUNOUT,'keys    dat',0) # open Key file on current drive
      write(LUNOUT,400) len # N
400   format(20(i3,1x))
      write(LUNOUT,400) (n(idx), idx=1,len)
      write(LUNOUT,400) len # D
      write(LUNOUT,400) (d(idx), idx=1,len)
      write(LUNOUT,400) lenp # P
      write(LUNOUT,400) (p(idx), idx=1,lenp)
      write(LUNOUT,400) lenq # Q
      write(LUNOUT,400) (q(idx), idx=1,lenq)
      endfile LUNOUT

end
##############################################################################

# GENPRM -- Generate a prime number

subroutine genprm(prime,lenp,work,modulo)

      byte prime(1),work(1),one,two,three,four,six,p0mod3,kadj
      logical prmtst
      integer lenp,modulo

      data one,two,three,four,six/1,2,3,4,6/

      len1=(2*lenp)/3 # get length of prime number needed to generate P or Q
      len2=lenp-len1 # get length of random number needed to generate P or Q
      call rmrkln('     Generating P0.')
      call genrnd(work,len1,modulo) # generate random number P0 to use as
                                    #  initial prime factor
      idx2=len1+1 # get index to start of K
      call rmrkln('     Generating K.')
```

```
call genrnd(work(idx2),len2,modulo) # generate random number K
if ((work(len1) & 1) == 0) # P0 even?
     work(len1)=work(len1)+1 # make P0 odd
if ((work(lenp) & 1) != 0) # K odd?
     work(lenp)=work(lenp)-1 # make K even
idxscr=lenp+1 # get index to work scratch area
lenscr=7*len1+3 # get length of work scratch area
call remark('      Making P0 Prime.')
while (! prmtst(work,len1,work(idxscr),lenscr,modulo)) # P0 not prime
     {
     call mpadd(work,len1,two,1,work(idxscr),len1+1,modulo)
                                                  # P0 = P0 + 2
     if (work(idxscr) >= 1) # P0 overflow, i.e. P0 = 10**(LEN1+1) + 1
          work(idxscr+1)=1 # keep P0 between 10**LEN1 & 10**(LEN1+1),
                           #  i.e. P0 --> 10**LEN1 + 1
     do idxo=1,len1 # move new P0 to old P0
          {
          idxn=idxscr+idxo # get index to new P0
          work(idxo)=work(idxn) # new P0 --> old P0
          }
     call rmrkln('.')
     call remark('                    .')

     }
do idxo=1,len1 # copy P0 to temporary P0 in preparation for division
     {
     idxn=idxscr+idxo # get index to temporary P0
     work(idxn)=work(idxo) # P0 --> temporary P0
     }
work(idxscr)=0 # make sure numerator has a leading zero
   idxquo=idxscr+len1+2 # get index to quotient
   lenquo=len1 # get length of quotient
   idxrem=idxquo+lenquo # get index to remainder
   call mpdiv(work(idxscr),len1+1,three,1,work(idxquo),lenquo,work(idxrem),
             1,modulo) # get P0 mod 3
  p0mod3=work(idxrem) # save P0 mod 3
do idx0=1,len2 # copy K to temporary K in preparation for division
        {
        idxo=idx2+idx0-1 # get index to K
        idxn=idxscr+idx0 # get index to temporary K
        work(idxn)=work(idxo) # K --> temporary K
        }
   work(idxscr)=0 # make sure numerator has a leading zero
   idxquo=idxscr+len2+2 # get index to quotient
   lenquo=len2 # get length of quotient
   idxrem=idxquo+lenquo # get index to remainder
   call mpdiv(work(idxscr),len2+1,three,1,work(idxquo),lenquo,work(idxrem),
             1,modulo) # get K mod 3
   if (p0mod3 != work(idxrem)) # P0 mod 3 != K mod 3?
        {
        # Note: N mod 3 = 0 or 2 or 1 (in that sequence)
        if (p0mod3 == 0) # generate K adjustment for P0 mod 3 == 0
             kadj=two*work(idxrem) # adjustment for K mod 3 == 2 or 1
        else if (p0mod3 == 2) # generate K adjustment for P0 mod 3 == 2
             kadj=two*(work(idxrem)+one) # adjustment for K mod 3 == 1 or 0
        else # generate K adjustment for P0 mod 3 == 1
             kadj=four-work(idxrem) # adjustment for K mod 3 == 0 or 2
        call mpadd(work(idx2),len2,kadj,1,work(idxscr),len2+1,modulo)
                          # force K mod 3 == P0 mod 3
        if (work(idxscr) >= 1) # K overflow, i.e. K = 10**(LEN2+1) + 0 or 2
```

(Continued on next page)

RSA (Listing Continued, text begins on page 30)

Listing Seven

```
          work(idxscr+1)=1 # keep K between 10**LEN2 & 10**(LEN2+1),
                           #  i.e. K --> 10**LEN2 + 0 or 2
     do idx0=1,len2 # move new K to old K
          {
          idxo=idx2+idx0-1 # get index to old K
          idxn=idxscr+idx0 # get index to new K
          work(idxo)=work(idxn) # new K --> old K
          }
     }
call mpmult(work,len1,work(idx2),len2,prime,lenp,modulo) # generate K * P0
lenscr=lenp+1 # get length of work scratch area
call mpadd(prime,lenp,one,1,work(idxscr),lenscr,modulo)
          # generate PRIME = K * P0 + 1 (10**LENP < PRIME < 10**(LENP+1))
do idxo=1,lenp # move new PRIME to old PRIME
     {
     idxn=idxscr+idxo # get index to new PRIME
     prime(idxo)=work(idxn) # new PRIME --> old PRIME
     }
# Note:  Keeping K mod 3 == P0 mod 3 just involves adding 6 to K
#        and (K + 6) * P0 + 1 == (K * P0 + 1) + (6 * P0)
#                             == PRIME + (6 * P0),
#        i.e. we are done with K !!!
idxscr=idx2 # get index to scratch work area
lenscr=len1+1 # get length of scratch area
call mpmult(work,len1,six,1,work(idxscr),lenscr,modulo) # generate 6 * P0
if (work(idxscr) != 0) # extra digit created?
     len1=len1+1 # get new length of 6 * P0
else # no extra digit created
     idxscr=idxscr+1 # advance index to (6 * P0) to skip over zero digit
     do idxo=1,len1 # move (6 * P0) to old P0
          {
          idxn=idxscr+idxo-1 # get index to (6 * P0)
          work(idxo)=work(idxn) # (6 * P0) --> old P0
          }
     idxscr=len1+1 # get index to scratch work area
     lenscr=lenp+1 # get length of work scratch area for adding (6 * P0)
     lenw=7*lenp+3 # get length of work scratch area for prime testing
     call rmrkln('.')
     call remark('     Generating the prime number.')
     while (! prmtst(prime,lenp,work(idxscr),lenw,modulo)) # PRIME not prime
          {
          call mpadd(prime,lenp,work,len1,work(idxscr),lenscr,modulo)
                    # PRIME = PRIME + (6 * P0)
          if (work(idxscr) >= 1) # PRIME overflow,
                                 #  i.e. PRIME = 10**(LENP+1) + ?
               work(idxscr+1)=1 # keep PRIME between 10**LENP & 10**(LENP+1),
                                #  i.e. PRIME --> 10**LENP + ?
          do idxo=1,lenp # move new PRIME to old PRIME
               {
               idxn=idxscr+idxo # get index to old PRIME
               prime(idxo)=work(idxn) # new PRIME --> old PRIME
               }
          call rmrkln('.')
          call remark('                                    .')
          }

     #
     # As a precaution at this point, may want to check that (PRIME + 1) does
     #  not consist entirely of rather small prime factors.  If so, should
     #  return to previous WHILE loop to get next PRIME and try again.
     #
     #    There are 168 primes (PRIMEk) less than 1000.  It is sufficient to
     #    show that if factoring PRIME by PRIMEk (k = 1,168) leaves a number
```

```
     #    which is greater than 10**5 then PRIME okay.
     #
     return

end
################################################################################

# GENRND -- Generate a random number of specified length

subroutine genrnd(rnd,lenr,modulo)

     byte rnd(1)
     integer lenr,modulo

     data rndnxt/.5/ # make RAN(*) generate the next random number
                     #  (any positive number will do)

     fmodul=float(modulo) # float byte modulus
     idxr=1 # get index to random number
     repeat # force 1st digit to be non-zero
          irnd=mod(int(fmodul*ran(rndnxt)),modulo) # get a random digit
     until (irnd != 0)
     rnd(idxr)=irnd # load 1st digit of RND
     idxr=idxr+1 # point to next digit of RND
     while (idxr <= lenr) # generate the rest of the random digits
          {
          irnd=mod(int(fmodul*ran(rndnxt)),modulo) # get a random digit
          rnd(idxr)=irnd # load random digit
          idxr=idxr+1 # advance pointer to next digit
          }
     return

end
```

End Listing Seven

Listing Eight

Copyright © 1983, Charles E. Burton, Denver, Colorado.
All rights reserved. Permission granted to use this software for personal, noncommercial purposes only.

```
# PROGRAM NAME:  EDCRYPT.RAT
# PURPOSE:  This routine uses the RSA (Rivest-Shamir-Adleman) keys generated
#           by GENPKEYS.RAT to Encrypt/Decrypt a message.
#           It uses the method described by D.E. Knuth, The Art of Computer
#           Programming, V. 2 (Semi-Numerical Algorithms), 2nd Ed., (Addison-
#           Wesley, Reading, MA), pp. 386-389.
#
# LANGUAGE:  RATFOR
# AUTHOR:  CEB
# USAGE:  ******** (main program)
#
#
#
# ARRAYS USED:  N(KEYLENGTH),D(KEYLENGTH),WORK(WORKLENGTH)
# EXTERNALS:  RPEXP
# UPDATE HISTORY:  INITIAL RELEASE -- 02/14/83 CEB
#

define(KEYLENGTH,90) # length of key for characters
define(WORKLENGTH,450) # length of WORK buffer for characters
```

(Continued on next page)

RSA (Listing Continued, text begins on page 30)

Listing Eight

```
                        #   (5 * KEYLENGTH)
define(BYTEMODULUS,128) # byte modulus for characters (1 character/byte)

define(LUNIN,6) # Input Logical Unit
define(LUNOUT,7) # Output Logical Unit
define(BLOCKS,8) # Number of 128 byte blocks to use for disk I/O

define(EOFCHAR,26) # text file EOF character (^Z)

program rsa

     character c,c0,getc,getch,oper,type,extn(3),linein(KEYLENGTH),
               lineot(KEYLENGTH)
     byte n(KEYLENGTH),d(KEYLENGTH),work(WORKLENGTH),three,msb
     integer openr,equal,getl
     double precision filnam # allow entry of up to 8 characters

     equivalence (filnam,work(1))

     data extn/BIGT,BIGX,BIGT/ # default file extension
     data modulo/BYTEMODULUS/ # get byte modulus for arithmetic
     data three/3/
     call rmrkln('.') # show options
     call rmrkln('Select operation:.')
     call rmrkln('      <E>ncryption.')
     call rmrkln('      <D>ecryption.')
     call rmrkln('.')
     repeat # get operation to perform
          {
          call remark('Operation -- .')
          oper=getc(c) & 95 # get selection and convert to upper case
          call rmrkln('.')
          }
     until ((oper == BIGE) | (oper == BIGD)) # valid entry
     call open(LUNIN,'keys    dat',0) # open key file on current drive
     read(LUNIN,100) lenn # get the length of N
100  format(i3)
     read(LUNIN,200) (n(idx), idx=1,lenn) # get N
200  format(20(i3,1x))
     if (oper == BIGD) # decryption selected?
          {
          read(LUNIN,100) lend # get length of D
          read(LUNIN,200) (d(idx), idx=1,lend) # get D (cube root of text)
          }
     else # encryption selected
          {
          lend=1 # only one digit
          d(lend)=three # set-up to cube text
          }
     endfile LUNIN # close key file
     repeat # get name of text file (.TXT extension assumed)
          {
          call rmrkln('.')
          if (oper == BIGD) # decryption selected?
               call remark('Encrypted.')
          else # encryption selected
               call remark('Clear.')
          call remark(' Text Source File Name (XXXXXXXX) -- .')
          read(CONSOLE,300) filnam # WORK(1 to 8)
300       format(a8)
```

```
        do idx=1,8 # convert lower case to upper case
             if (type(work(idx)) == LETTER) # is character a letter?
                  work(idx)=work(idx) & 95 # to upper case
        work(9)=EOS # terminate file name
        }
   until (openr(LUNIN,READONLY,BLOCKS,work,extn) == YES) # successful open
   call scopy(work,1,work,10) # save source file name
   repeat # get name of text file (.TXT extension assumed)
        {
        call rmrkln('.')
        if (oper == BIGD) # decryption selected?
             call remark('Clear.')
        else # encryption selected
             call remark('Encrypted.')
        call remark(' Text Destination File Name (XXXXXXXX) -- .')
        read(CONSOLE,300) filnam # WORK(1 to 8)
    do idx=1,8 # convert lower case to upper case
         if (type(work(idx)) == LETTER) # is character a letter?
              work(idx)=work(idx) & 95 # to upper case
    work(9)=EOS # terminate file name
    if (equal(work,work(10)) == YES) # Source & Destination names same?
         {
         call rmrkln('.')
         call remark('Source and Destination names cannot be the .')
         call rmrkln('same !!!.')
         }
    }
until (openr(LUNOUT,WRITEONLY,BLOCKS,work,extn) == YES) # successful open
for (c0=0; c0 != EOF; ) # convert text
    {
    if (oper == BIGE) # encryption?
         {
         linein(1)=0 # make sure LINEIN mod N = LINEIN
         len=2 # point to next character
         }
    else # decryption
         len=1 # initialize line length counter
    repeat # get a line of text (encrypted or decrypted)
         {
         linein(len)=getch(c0,LUNIN) & 127 # get a character/strip MSBit
                 # (no effect on Clear Text but MSBit set by encryption)
         len=len+1 # point to next character position
         if ((oper == BIGE) & (c0 == EOF)) # encryption & EOF?
              {
              linein(len-1)=EOFCHAR # overwrite EOF with EOFCHAR
              while (len <= lenn) # clear text line
                   {
                   repeat
                        num=int(32.0*ran(.5)) # get a number (0 - 31)
                   until (num != EOFCHAR) # not text EOF character
                   linein(len)=num # pad line with a random character
                                   #  (could have previously seeded
                                   #  random number generator !!!)
                   len=len+1 # advance line length counter
                   }
              }
         }
    until ((c0 == EOF) | (len > lenn)) # ready to encrypt/decrypt
    if ((oper == BIGD) & (c0 == EOF)) # decryption & EOF?
         break # encryption does not leave characters with EOF, so done
    lenin=len-1 # point to last actual character
    lenout=lenn # get length of converted line
    lenw=4*lenn+3 # get length of work area
    call rpexp(linein,lenin,d,lend,lineot,lenout,n,lenn,work,lenw,modulo)
                  # encrypt/decrypt line

  if (oper == BIGD) # decryption selected?
```

(Continued on next page)

Listing Eight

```
                {
                len0=2 # skip over leading zero from encryption
                msb=0 # MSBit for decrypted text is zero
                if (c0 == EOF) # end of file found on input?
                     {
                     while ((lineot(lenout) != EOFCHAR) & (lenout >= len0))
                                                  # strip pad characters
                          lenout=lenout-1 # backup end of line pointer
                     lenout=lenout-1 # backup over EOFCHAR
                     }
                }
           else # encryption
                {
                len0=1 # use all characters
                msb=128 # MSBit for encrypted text is one for Control Chars.
                }
           if (lenout >= len0) # at least one character to output?
                {
                do idx=len0,lenout # save encrypted/decrypted text
                     {
                     c=lineot(idx) # get character to save
                     if (c < BLANK) # Control Character (only used during
                                    #  encryption)?
                          c=c | msb # move Control Characters above printable
                                    #  ASCII range, especially EOFCHAR !!!
                     call putch(c,LUNOUT)
                     }
                }
           }
      call closer(LUNIN) # close files
      call closer(LUNOUT)

end
```

End Listing Eight

Listing Nine

```
#     Random Number Generator for use with APULIB
#
#          Modeled after W.J. Cody & W. Waite, "Software Manual for the
#               Elementary Functions"
#

real function ran(x)

      real x,y,const,zero

      data y/100001.0/,const/2796203.0/,zero/0.0/

      if (x == zero)
           ran=y
      else
           {
           if (x < zero)
                y=-x*const
           y=amod(125.0*y,const)
           ran=y/const
           if (y == zero)
                y=100001.0
           }
      return

end
```

End Listings

BASICFMT for TRS-80

[A BASIC Formatter]

OK, programmer! It's time to play "Find the Variable"! The object of the game is to determine what the variables are in line 10 of my program. The winner gets a Rubik's Cube!

```
10  IFBASICANDCOMPRESSEDTHEN
SANITY=CRAZYANDYESELSEIFNOT
BASICANDCOMPRESSEDTHENSANIT
Y=CRAZYANDNO
```


by Davy Crockett

Davy Crockett, 5807 Cherrywood Ln., Apt. 104, Greenbelt, MD 20770

The Game

Many programmers like to play games with their programs. I do not mean after you type RUN. These games are played when RUN would only elicit a READY prompt from the BASIC interpreter. They choose variable names like DO. That way, when you use it in an IF statement it can look like this:

```
100 IF DOOR KNOB THEN 300
```

Of course, judicious use of spacing could clarify the issue, but that is contrary to the rules of the game.

Now I know what you are thinking: "This guy is leading up to yet another program to format a BASIC ASCII file." You are correct on one count: this is a formatting program. However, it is not just another program. BASICFMT reads a BASIC program in *compressed* format from disk and prints a readable listing. In order to use BASICFMT, all you need is a BASIC program stored on disk without using the ",A" option. No longer is it necessary to load the program and then save it again with the ASCII option.

The Game Pieces

The first byte of these BASIC files is always 0FFH. This byte tells BASIC that it is all right to continue loading this file as a BASIC program. If this byte is not 0FFH then you get the old "Direct Statement in File" error from BASIC. Following this byte, each line is stored in

```
10 '
       **********************  Version 1.0
       *****  DEMO/BAS  *****  29 October 1982
       **********************  Davy * Crockett

1510 '
       Davy * Crockett, B.S., Esquire
       5807 Cherrywood Lane, Apt. 104
       Greenbelt, Maryland      20770

3010 '
       Program to demonstrate BASICFMT

4510 CLS:CLEAR1000:DEFINTA-Z:DIMNM$(35),AD$(35),TL$(2):GOSUB13510:ONER
RORGOTO12010
6010 DEFFNLN$=LEFT$(NM$(I),INSTR(1,NM$(I),",")-1)
7510 GOTO10510:REM - remark statement
9010 CLS:PRINTTL$(1);:PRINTTL$(2):RETURN
10510 GOSUB9010:I=35:PRINT"Please enter your name (last, first)":INPUT
NM$(I),NM$(I-1):NM$(I)=NM$(I)+", "+NM$(I-1):LN$=FNLN$:PRINT:PRINT"Hell
o, ";LN$:FORK=0TO200:NEXT:GOTO10510
12010 PRINT"ERL =";ERL;"   ERR =";ERR:ONERRORGOTO0
13510 FORK=1TO2:FORM=1TO21:READY:TL$(K)=TL$(K)+CHR$(Y):NEXT:NEXT:RETUR
N
15010 DATA191,131,171,148,32,191,179,179,131,32,191,140,176,140,191,32
,191,131,131,191,10,143,140,142,129,32,143,140,140,140,32,143,32,32,32
,143,32,143,140,140,143,10
```

Figure 1.

BASIC'S LLIST command

a specific format.

The first two bytes of each line contain an address which points to the line number of the following line. The second pair of bytes is the line number in binary format. After this is the actual statement with all of the reserved words compressed into a single byte "token." A binary zero (00H) is the end-of-line marker. For example, the preceding line 100 would, in compressed format on disk, appear as in Figure 1 (page 60). The end of the program is indicated by a next-line pointer consisting of two binary zeros.

Playing The Game

The BASIC tokens are all hex codes which cannot be generated by simply pressing keys on the keyboard. This makes them easy to identify in a section of compressed BASIC code. The character strings associated with each token are stored in the ROM at location 1650H for both the Model I and the Model III. Once a token is found, it is a simple matter to extract the character string associated with that token from the table stored in the ROM and separate it by blanks before printing.

The first thing I noticed is that BASIC does not always compress a program the way I expected it to, particularly the "remark" commands. The characters REM get translated to a token (93H) as you would expect. However, when I used an apostrophe to indicate a remark, three bytes were inserted instead of a single byte token. The first byte was a colon, the second byte was the remark token, and the third byte was the hex value 0FBH. The 0FBH apparently tells BASIC to back up to the preceding colon and replace it with an apostrophe when listing the program.

The ELSE command contained in an IF statement is another reserved word that is not treated as you would expect. The ELSE command is always converted to a two-byte sequence with a colon preceding the ELSE token.

Winning At The Game

BASICFMT can help beat these gamesters at their own game by separating the variables and the command words by blanks. The program opens by asking for the filespec of the BASIC file to be formatted. It will then open the file and check to make sure that it is a BASIC program in compressed format. If the file is not in the expected format, an error message is printed and you are returned to the DOS.

When the file checks out, you will be asked for printer specifications. These include: maximum line length, number of lines to print on a page, and number of lines to skip between pages. Printing begins immediately upon entry of the last printer specification. If the printer is not ready, the program will print a mes-

```
   10 '
              **********************  Version 1.0
              *****  DEMO/BAS  *****  29 October 1982
              **********************  Davy * Crockett

 1510 '
              Davy * Crockett, B.S., Esquire
              5807 Cherrywood Lane, Apt. 104
              Greenbelt, Maryland      20770

 3010 '
              Program to demonstrate BASICFMT

 4510 CLS : CLEAR 1000 : DEFINT A - Z : DIM NM$(35), AD$(35), TL$(2)
      : GOSUB 13510 : ON ERROR GOTO 12010
 6010 DEF FN LN$ = LEFT$ (NM$(I), INSTR (1, NM$(I), ",") - 1)
 7510 GOTO 10510 : REM - remark statement
 9010 CLS : PRINT TL$(1); : PRINT TL$(2) : RETURN
10510 GOSUB 9010 : I = 35 : PRINT "Please enter your name (last,
      first)" : INPUT NM$(I), NM$(I - 1) : NM$(I) = NM$(I) + ", " +
      NM$(I - 1) : LN$ = FN LN$ : PRINT : PRINT "Hello, "; LN$ : FOR
      K = 0 TO 200 : NEXT : GOTO 10510
12010 PRINT "ERL ="; ERL ; "   ERR ="; ERR : ON ERROR GOTO 0
13510 FOR K = 1 TO 2 : FOR M = 1 TO 21 : READ Y : TL$(K) = TL$(K) +
      CHR$ (Y) : NEXT : NEXT : RETURN
15010 DATA 191, 131, 171, 148, 32, 191, 179, 179, 131, 32, 191, 140,
      176, 140, 191, 32, 191, 131, 131, 191, 10, 143, 140, 142, 129,
      32, 143, 140, 140, 140, 32, 143, 32, 32, 32, 143, 32, 143, 140,
      140, 143, 10
```

Figure 2.

BASICFMT output

sage for you to ready the printer and then wait for you to press ENTER.

The target program is read by a routine (GETBYT) which returns a single byte from the disk buffer each time it is called. Some programming languages, notably PL/I, call this technique "character stream" input. One line at a time is read this way, decompressed, and stored in a primary buffer. Extra blanks separating the command words, colons, semi colons, and commas are inserted in the statement at this point. All remark statements and things enclosed within quotation marks are not changed in any way.

Phase two consists of moving the contents of the primary buffer to a secondary buffer. While this is being done, the line breaks are inserted in the statement. Each line is broken at a blank if possible. This would not be possible if there were a remark or quoted string which had no blanks and completely filled the last 10 characters on the line. In this case, the line is broken at the maximum line length specified. Subsequent lines of the same statement are indented past the line number to make the line number easy to read.

Finally, each line in the secondary buffer is printed. The page length is checked before each line to make sure it will not exceed the maximum number of lines specified as page length. Processing continues with the next line until the end of the file is reached. You are then returned to the DOS READY prompt.

Game Strategy

This program will run under any DOS which uses the standard entry points to open an existing file (4424H), read a sector (4436H), and close a file (4428H). It will also run on either the Model I or the Model III. This is due to several factors. First, the BASIC token table is located at the same address in ROM for both models. Second, I used the assembler instruction LD A,(37E8H) to check the printer status instead of IN A,(0FBH), which only works on the Model III. Third, many of the DOS/ROM routines are incorporated in the program so that you do not have to worry about them when switching from one DOS to another. These include keyboard input, screen and printer output, ASCII characters to binary value, binary value to ASCII decimal characters, and even a routine to simulate the single-byte read from disk available with NEWDOS/80.

Some of you may wish to add a routine that prompts for a title line to be printed at the top of each page. I did not feel that this was necessary because I always include an identification section for each program at the beginning. You could also modify the PAGER routine to print a page number when it ejects a page. If your printer requires something separating carriage returns or line feed characters, the LFEED routine will have to be modified to transmit the required codes.

Before I forget, this statement may be easier to read:

```
10 IF BASIC AND COMPRESSED
   THEN SANITY = CRAZY AND YES
   ELSE IF NOT BASIC AND
   COMPRESSED THEN SANITY =
   CRAZY AND NO
```

or in other words:

```
10 IF BA AND CO THEN SA = CR AND
   YE ELSE IF NOT BA AND CO
   THEN SA = CR AND NO
```

■■J

(Listing begins on next page)

BASICFMT (Text begins on page 60)

```
00010 ;=======================================================*
00020 ;                                                       *
00030 ;       **********************  Version 3.4             *
00040 ;       *****  BASICFMT  *****  01 November 1983        *
00050 ;       **********************  Davy * Crockett         *
00060 ;                                                       *
00070 ;       Davy * Crockett, B.S., Esquire                  *
00080 ;       5807 Cherrywood Lane, Apt. 104                  *
00090 ;       Greenbelt, Maryland      20770                  *
00100 ;                                                       *
00110 ;       This program will print a formatted listing     *
00120 ;       of a basic program from disk.  The program      *
00130 ;       must not, I repeat "NOT", be stored in ascii    *
00140 ;       format.  BASICFMT prompts for basic filespec,   *
00150 ;       printer line length, lines per page, and        *
00160 ;       spacing between pages for the output.           *
00170 ;                                                       *
00180 ;=======================================================*
00190 ;
00200 ;>>==>  System equates section
00210 ;
00220 TOKTAB  EQU     1650H   ;basic's token table address
00230 LPPORT  EQU     37E8H   ;line printer status port
00240 CTLKEY  EQU     3840H   ;control key address
00250 OPEN    EQU     4424H   ;open existing disk file
00260 READ    EQU     4436H   ;read a record from disk
00270 CLOSE   EQU     4428H   ;close a disk file
00280 DOSERR  EQU     4409H   ;display DOS error
00290 DOS     EQU     402DH   ;return to operating system
00300 ;
00310         ORG     5800H
00320 ;
00330 BASFMT  LD      SP,BASFMT       ;initialize stack
00340 GETFS   LD      DE,FSMSG        ;point to prompt
00350         LD      HL,DCB          ;point to FCB
00360         LD      B,32            ;maximum chars
00370         CALL    INPUTP          ;get filespec
00380         CP      1               ;<BREAK>?
00390         JP      Z,DOS           ;yes, exit
00400         LD      DE,DCB          ;point to FCB
00410         LD      HL,BUFFER       ;point to buffer
00420         LD      B,0H            ;LRECL=256
00430         CALL    OPEN            ;open the file
00440         JP      NZ,DOSERR       ;error, exit
00450         CALL    GETBYT          ;get a byte
00460         CP      0FFH            ;BASIC file?
00470         JR      Z,GETLL         ;yes, continue
00480         LD      HL,FYLMSG       ;no, point to error msg
00490         CALL    PRINT           ; and display message
00500         JR      GETFS           ;return to get filespec
00510 ;
00520 ;>>==>  Get printer characteristics from operator
00530 ;
00540 GETLL   LD      DE,LLMSG        ;point to prompt
00550         CALL    GETVAL          ;get line length
00560         JR      Z,GETPL         ;got something?
00570         SUB     6               ;adjust line length
00580         LD      (LL),A          ; and store it
00590 GETPL   LD      DE,PLMSG        ;point to prompt
00600         CALL    GETVAL          ;get page length
```

```
00610           JR      Z,GETPS         ;got something?
00620           LD      (PL),A          ;store page length
00630 GETPS     LD      DE,PSMSG        ;point to prompt
00640           CALL    GETVAL          ;get page spacing
00650           JP      Z,BEGIN         ;got something?
00660           LD      (PS),A          ;yes, store it
00670           JP      BEGIN           ; and start
00680 ;
00690 ;>>==>  Obtain a decimal value from the keyboard
00700 ;       and convert it to a binary value.
00710 ;
00720 GETVAL    LD      HL,LINE         ;point to buffer
00730           LD      B,4             ;maximum chars
00740           CALL    INPUTP          ;get characters
00750           CP      1H              ;<BREAK>?
00760           JR      Z,EXIT          ;yes, exit
00770           CALL    DECBIN          ;convert to binary
00780           LD      A,E             ;get binary value
00790           OR      A               ;set/reset Z flag
00800           RET
00810 ;
00820 DONE      LD      A,0CH           ;form feed
00830           CALL    LPBYTE          ;eject page
00840 EXIT      LD      DE,DCB          ;point to FCB
00850           CALL    CLOSE           ;close the file
00860           JP      DOS             ;exit, stage left
00870 ;
00880 ;>>==>  Prompt and information messages section
00890 ;
00900 FSMSG     DEFW    1F1CH           ;home & cls
00910           DEFM    '=*= Basicfmt =*= by Davy * Crockett'
00920           DEFW    0D0DH           ;couple a <CR>'s
00930           DEFM    'Enter source filespec'
00940           NOP
00950 LLMSG     DEFB    0DH
00960           DEFM    '* File successfully opened'
00970           DEFW    0D0DH
00980           DEFM    'Enter line length (<ENTER>=85)'
00990           NOP
01000 PLMSG     DEFB    0DH
01010           DEFM    'Enter page length (<ENTER>=60)'
01020           NOP
01030 PSMSG     DEFB    0DH
01040           DEFM    'Enter page spacing (<ENTER>=6)'
01050           NOP
01060 FYLMSG    DEFB    0DH
01070           DEFM    '>>==> File not compressed BASIC'
01080           NOP
01090 ;
01100 ;         **********************************
01110 ;         * Main program control section. *
01120 ;         **********************************
01130 ;
01140 BEGIN     CALL    LPSTAT          ;check printer status
01150 NXTLYN    CALL    GETLYN          ;buffer a statement
01160           CALL    PUTLYN          ;print a statement
01170           JR      NXTLYN          ;go get another
01180 ;
01190 ;>>==>  Store a statement number in the primary buffer
01200 ;
01210 GETLYN    CALL    GETBYT          ;get a byte
01220           LD      L,A             ; and store in (L)
01230           CALL    GETBYT          ;get another byte
01240           LD      H,A             ; and store in (H)
```

(Continued on next page)

```
01250           LD      DE,0            ;test value
01260           CALL    CPHLDE          ;zero address?
01270           JP      Z,DONE          ;yes, quit
01280           LD      DE,LINE         ;point to buffer
01290           CALL    GETBYT          ;LSB line number
01300           LD      L,A             ; and store in (L)
01310           CALL    GETBYT          ;MSB line number
01320           LD      H,A             ; and store in (H)
01330           CALL    BINDEC          ;convert to decimal
01340           LD      DE,LINE+5       ;end of line #
01350           LD      A,' '           ;get a blank
01360           LD      (DE),A          ; and store after #
01370           XOR     A               ;set the quote
01380           LD      (QUOTE),A       ; indicator
01390           LD      (REMARK),A      ; and rem indicator
01400 ;
01410 ;>>==>    Store the rest of the statement in the buffer
01420 ;
01430 BUILD     CALL    GETBYT          ;get a byte
01440           OR      A               ;end of the line?
01450           RET     Z               ;yes, return
01460           CP      80H             ;token?
01470           JR      NC,TOKEN        ;yes, jump
01480 CHK22     CP      '"'             ;quote?
01490           JR      NZ,CHK10        ;no, continue
01500           PUSH    AF              ;save (AF)
01510           LD      A,(QUOTE)       ;get quote value
01520           XOR     1               ;switch it
01530           LD      (QUOTE),A       ; and put it back
01540           POP     AF              ;restore (AF)
01550           JR      CONT            ; and continue
01560 CHK10     CP      0AH             ;line feed?
01570           JR      NZ,VALID        ;no, check validity
01580           LD      A,0DH           ;replace w/ <CR>
01590           JR      CONT            ; and continue
01600 VALID     CP      ' '             ;valid char?
01610           JR      C,BLANK         ;no, zap it
01620           CP      ':'             ;colon character?
01630           CALL    Z,SPACE         ;yes, add space
01640 CONT      INC     DE              ;bump pointer
01650           LD      (DE),A          ;store character
01660           CP      ','             ;comma?
01670           JR      Z,CONT1         ;yes, insert blank
01680           CP      ';'             ;semi-colon?
01690           JR      Z,CONT1         ;yes, insert blank
01700           CP      ':'             ;colon character?
01710 CONT1     CALL    Z,SPACE         ;yes, add a space
01720 CONT2     JR      BUILD           ;go for next
01730 BLANK     LD      A,' '           ;zap the char
01740           JR      CONT            ; and continue
01750 ;
01760 ;>>==>    Obtain characters represented by basic token
01770 ;         from the token table in the basic ROM.
01780 ;
01790 TOKEN     CP      0FCH            ;garbage char?
01800           JR      NC,BLANK        ;yes, zap it
01810           PUSH    AF              ;save token
01820 CHK0FB    CP      0FBH            ;zap REM token?
01830           JR      NZ,CHK95        ;no, continue
01840           DEC     DE              ;backup
01850           DEC     DE              ; to the
01860           DEC     DE              ;  prior
01870           DEC     DE              ;   colon
```

```
01880           LD      A,27H           ;change colon
01890           LD      (DE),A          ; to quote (')
01900           POP     AF              ;clear the stack
01910           JR      BUILD           ;go get next character
01920 CHK95     CP      95H             ;ELSE token?
01930           JR      NZ,CHK93        ;no, continue
01940           DEC     DE              ;yes, delete the prior
01950           DEC     DE              ;  space and colon
01960           JR      OFFSET          ;   and continue
01970 CHK93     CP      93H             ;REM token?
01980           JR      NZ,OFFSET       ;no, compute offset
01990           LD      (REMARK),A      ;set REM indicator
02000 OFFSET    SUB     7FH             ;get position in table
02010           LD      B,A             ; and put in (B)
02020           LD      HL,TOKTAB       ;point to token table
02030 TOKEN1    LD      A,(HL)          ;get table character
02040           INC     HL              ;increment the pointer
02050           CP      7FH             ;first character of a token?
02060           JR      C,TOKEN1        ;no, get next character
02070           DJNZ    TOKEN1          ;yes, loop until token
02080           SUB     80H             ;got it, adjust first char
02090           CALL    SPACE           ;insert space (maybe)
02100 TOKEN2    INC     DE              ;increment buffer pointer
02110           LD      (DE),A          ;and store in buffer
02120           LD      A,(HL)          ;get next character
02130           INC     HL              ;increment token pointer
02140           CP      7FH             ;end of token?
02150           JR      C,TOKEN2        ;no, continue moving token
02160           POP     AF              ;retrieve token
02170           CP      0BCH            ;TAB( token?
02180           JP      Z,BUILD         ;yes, don't put blank
02190           CP      93H             ;REM token?
02200           JP      Z,BUILD         ;yes, don't put blank
02210           JR      BLANK           ;no, insert space
02220 ;
02230 ;>>==>  Conditional blank insert routine
02240 ;       Inserts a blank in the output buffer if:
02250 ;       1)  we are not processing a REM statement, and
02260 ;       2)  we are not processing within quotes,   and
02270 ;       3)  there is not a blank there all ready.
02280 ;
02290 SPACE     PUSH    AF              ;save (AF)
02300           LD      A,(QUOTE)       ;get quote value
02310           OR      A               ;inside quotes?
02320           JR      NZ,XSPACE       ;yes, no insert
02330           LD      A,(REMARK)      ;get rem value
02340           OR      A               ;inside remark?
02350           JR      NZ,XSPACE       ;yes, no insert
02360           LD      A,(DE)          ;get last char
02370           CP      ' '             ;is it a blank?
02380           JR      Z,XSPACE        ;yes, ok
02390           INC     DE              ;bump pointer
02400           LD      A,' '           ;get a blank
02410           LD      (DE),A          ;and add it
02420 XSPACE    POP     AF              ;retrieve (AF)
02430           RET
02440 ;
02450 ;>>==>  At this point, the entire statement is in the
02460 ;       primary output buffer.  This section moves the
02470 ;       statement to the secondary output buffer and
02480 ;       inserts line breaks and indentation as required.
02490 ;
```

(Continued on next page)

```
02500 PUTLYN  XOR     A               ;zero (A)
02510         INC     DE              ;mark the end
02520         LD      (DE),A          ; of the statement
02530         LD      DE,OUTLYN       ;set output pointer
02540         LD      HL,LINE         ;set input pointer
02550         LD      A,(LL)          ;get max line length
02560         ADD     A,6             ;add length of #
02570         LD      B,A             ;and put in (B)
02580 PUTNXT  LD      A,(HL)          ;get a byte
02590         LD      (DE),A          ; and store it
02600         INC     HL              ;bump in pointer
02610         INC     DE              ;bump out pointer
02620         CP      0DH             ;<LF>?
02630         JR      Z,SPACER        ;yes, insert indent
02640         OR      A               ;end of input?
02650         JR      Z,ENDLYN        ;yes, jump out
02660         DJNZ    PUTNXT          ;no, continue
02670 ;
02680 ;>>==>  This section will backup and insert a line break
02690 ;       in the secondary buffer when the maximum line
02700 ;       length has been reached.  If a blank is not found
02710 ;       in the last 10 characters stored, the line break
02720 ;       is inserted at the maximum line length location.
02730 ;
02740 BACKUP  LD      (OLDDE),DE      ;save current (DE)
02750         LD      (OLDHL),HL      ;and current (HL)
02760         LD      B,0AH           ;limiter value
02770 BACK    DEC     HL              ;unbump pointer
02780         DEC     DE              ;unbump pointer
02790         LD      A,(HL)          ;get character
02800         CP      ' '             ;blank?
02810         JR      Z,BREAK         ;yes, break here
02820         DJNZ    BACK            ;no, try another
02830         LD      HL,(OLDHL)      ;can't break it
02840         LD      DE,(OLDDE)      ; use old breaks
02850         DEC     HL              ;unbump for bump
02860         DEC     DE              ;unbump for bump
02870 BREAK   INC     HL              ;skip blank
02880         INC     DE              ;skip blank
02890         LD      A,0DH           ;insert a <CR>
02900         LD      (DE),A          ; in the line
02910         INC     DE              ;bump pointer
02920 SPACER  LD      A,' '           ;get a blank
02930         LD      B,6             ; and blank count
02940 SPACE2  LD      (DE),A          ;insert a blank
02950         INC     DE              ;bump pointer
02960         DJNZ    SPACE2          ;next?
02970         LD      A,(LL)          ;reset the line
02980         LD      B,A             ; length value
02990         JR      PUTNXT          ;go back for more
03000 OLDHL   DEFW    0               ;storage space
03010 OLDDE   DEFW    0               ;storage space
03020 ;
03030 ;>>==>  The statement has now been stored in the secondary
03040 ;       buffer with line breaks and indentation.  ENDLYN
03050 ;       prints the secondary buffer on the line printer
03060 ;       inserting page eject commands as necessary.
03070 ;
03080 ENDLYN  DEC     DE              ;back up one
03090         LD      A,0DH           ;get a <CR>
03100         LD      (DE),A          ; and insert it
03110         INC     DE              ;bump pointer
```

```
03120           XOR     A               ;get a EOL char
03130           LD      (DE),A          ; and mark it
03140           LD      HL,OUTLYN       ;point to output
03150 NEXTLN    CALL    PAGER           ;check page break
03160           CALL    LPRINT          ;print a line
03170           INC     HL              ;bump over <CR>
03180           LD      A,(HL)          ;get next char
03190           OR      A               ;EOL?
03200           JR      NZ,NEXTLN       ;no, get more
03210           RET                     ;all done
03220 ;
03230 ;>>==>    This routine generates carriage returns to
03240 ;         eject a page when the page is full.
03250 ;
03260 PAGER     LD      A,(LINES)       ;get line count
03270           INC     A               ; and add 1
03280           LD      (LINES),A       ; and store back
03290           LD      B,A             ;store in (B)
03300           LD      A,(PL)          ;get page length
03310           CP      B               ;too many lines?
03320           RET     NC              ;no, return
03330           LD      A,1             ;yes, repage
03340           LD      (LINES),A       ;reset counter
03350           LD      A,(PS)          ;get page spacing
03360           LD      B,A             ; and store in (B)
03370           LD      A,0DH           ;<LF> value
03380 LFEED     CALL    LPBYTE          ;feed a line
03390           DJNZ    LFEED           ;more?
03400           RET                     ;back for more
03410 ;
03420 ;>>==>    This routine returns a single byte from the
03430 ;         disk buffer to the caller in register A.
03440 ;
03450 GETBYT    PUSH    HL              ;save (HL)
03460           PUSH    DE              ;save (DE)
03470           LD      HL,(CHRPTR)     ;get byte pointer
03480           LD      DE,ENDBUF       ;and end of buffer
03490           INC     HL              ;bump pointer
03500           CALL    CPHLDE          ;compare them
03510           JR      Z,GREAD         ;need next sector
03520 GET02     LD      A,(CTLKEY)      ;get control key
03530           BIT     2,A             ;<BREAK>?
03540           JP      NZ,EXIT         ;yes, stop
03550           LD      A,(HL)          ;get the byte
03560           LD      (CHRPTR),HL     ;store pointer
03570           POP     DE              ;retrieve (DE)
03580           POP     HL              ;retrieve (HL)
03590           RET                     ;send back byte
03600 ;         storage for address of last character returned
03610 CHRPTR    DEFW    ENDBUF-1
03620 ;         read next sector of the basic program
03630 GREAD     LD      DE,DCB          ;point to FCB
03640           CALL    READ            ;read a sector
03650           LD      HL,BUFFER       ;point to buffer
03660           JR      Z,GET02         ; and return byte
03670           OR      80H             ;set return code
03680           CALL    DOSERR          ;display error
03690           JP      EXIT            ;exit, stage left
03700 ;
03710 ;=================================================
03720 ;  BINDEC  -  (HL)=binary  -  (DE)-->output buffer
03730 ;  The output buffer must be exactly 5 bytes long.
03740 ;=================================================
03750 ;
03760 BINDEC    LD      (BINA),HL       ;save binary number
03770           LD      (BIND),DE       ;save beginning of buffer
```

(Continued on next page)

```
03780           INC     DE              ;increment (DE)
03790           INC     DE              ; to the end
03800           INC     DE              ;  of the
03810           INC     DE              ;   buffer
03820           LD      (BIN4),DE       ;store end of buffer address
03830           LD      B,5H            ;number of bytes in buffer
03840           XOR     A               ;initial value for buffer
03850           LD      HL,(BIND)       ;beginning of buffer
03860 BD1       LD      (HL),A          ;initialize the
03870           INC     HL              ; buffer to all
03880           DJNZ    BD1             ;  binary zeros
03890           LD      HL,(BINA)       ;retrieve the binary number
03900           LD      A,H             ;get the MSB
03910           CALL    BD4             ; and convert to decimal
03920           LD      A,(BINA)        ;get the LSB
03930           CALL    BD4             ; and convert to decimal
03940           LD      HL,(BIND)       ;buffer beginning address
03950           LD      B,5H            ;buffer length
03960 BD2       LD      A,30H           ;convert to ascii value
03970           ADD     A,(HL)          ;convert digit to ascii
03980           LD      (HL),A          ; and store back in buffer
03990           INC     HL              ;increment pointer
04000           DJNZ    BD2             ;all digits done?
04010           LD      HL,(BIND)       ;beginning of buffer
04020           LD      B,4H            ;buffer length less 1
04030 BD3       LD      A,30H           ;ascii zero
04040           CP      (HL)            ;is buffer char "0"
04050           RET     NZ              ;no, return to caller
04060           LD      A,20H           ;yes, change to a blank
04070           LD      (HL),A          ; and store in buffer
04080           INC     HL              ;increment pointer
04090           DJNZ    BD3             ;loop for all but last char
04100           RET                     ;return to calling routine
04110 BINA      DEFW    0H              ;storage for binary value
04120 BIND      DEFW    0H              ;storage beginning of buffer
04130 BIN4      DEFW    0H              ;storage end of buffer
04140 BD4       LD      B,8H            ;bits per byte
04150 BD5       PUSH    BC              ;save loop counter
04160           PUSH    AF              ;save (AF)
04170           CALL    BD6             ;multiply buffer by 2
04180           POP     AF              ;retrieve the value
04190           RLA                     ;shift bit to carry flag
04200           PUSH    AF              ;save the value again
04210           CALL    C,BD9           ;if bit set, add to buffer
04220           POP     AF              ;retrieve value again
04230           POP     BC              ;restore loop counter
04240           DJNZ    BD5             ;continue for all bits
04250           RET                     ;return to caller
04260 ;
04270 ;>>==>  Multiply the contents of the buffer by 2
04280 ;
04290 BD6       LD      HL,(BIN4)       ;end of buffer address
04300           OR      A               ;reset the carry flag
04310           LD      B,5H            ;length of buffer
04320 BD7       LD      A,(HL)          ;get a digit
04330           ADC     A,A             ;multiply by 2 and add carry
04340           CP      0AH             ;less than 10?
04350           JR      C,BD8           ;yes, skip adjustment
04360           SUB     0AH             ;adjust to less than 10
04370 BD8       LD      (HL),A          ;return to buffer
04380           CCF                     ;switch the carry flag
```

```
04390           DEC     HL              ;point to next digit
04400           DJNZ    BD7             ;loop for all digits
04410           RET                     ;return to caller
04420 ;
04430 ;>>==>  Add 1 to the contents of the buffer
04440 ;
04450 BD9       LD      HL,(BIN4)       ;end of buffer address
04460           LD      B,5H            ;length of the buffer
04470 BDA       LD      A,(HL)          ;get a digit
04480           INC     A               ;add 1 to the digit
04490           LD      (HL),A          ; and return it to buffer
04500           CP      0AH             ;digit less than 10?
04510           RET     C               ;yes, return to caller
04520           SUB     0AH             ;no, adjust to decimal
04530           LD      (HL),A          ; and return to buffer
04540           DEC     HL              ;point to next digit
04550           DJNZ    BDA             ;loop to handle carry
04560           RET                     ;return to caller
04570 ;
04580 ;=== DECBIN === Calls: nothing
04590 ;=============================================*
04600 ;======  Convert ascii decimal to binary  ======*
04610 ;=============================================*
04620 ;                                              *
04630 ;  on entry:  (HL) => character string         *
04640 ;             terminated by first non-decimal  *
04650 ;             character encountered            *
04660 ;                                              *
04670 ;  on exit:   (DE) =  binary equivalent        *
04680 ;             (HL) => non-decimal character    *
04690 ;                                              *
04700 ;=============================================*
04710 ;
04720 DECBIN  DEC     HL              ;set initial pointer
04730         LD      DE,0H           ;zero the contents DE
04740 DB1     INC     HL              ;point to next character
04750         LD      A,(HL)          ;get next character
04760         SUB     30H             ;adjust to binary
04770         RET     C               ;return if character < "0"
04780         CP      0AH             ;character greater than 9
04790         RET     NC              ;yes, return to caller
04800         PUSH    HL              ;save character pointer
04810         LD      H,D             ;get current value
04820         LD      L,E             ; from DE register
04830         ADD     HL,HL           ;old value times 2
04840         ADD     HL,HL           ;old value times 4
04850         ADD     HL,DE           ;old value times 5
04860         ADD     HL,HL           ;old value times 10
04870         LD      E,A             ;value of new digit
04880         LD      D,0H            ; to DE register
04890         ADD     HL,DE           ;add to the old value
04900         EX      DE,HL           ; and return value to DE
04910         POP     HL              ;retrieve character pointer
04920         JR      DB1             ;go to process next char
04930 ;
04940 ;=== PRINT === Calls: OUTBYT
04950 ;==============================*
04960 ;  Output a line to the screen  *
04970 ;==============================*
04980 ;
04990 PRINT   PUSH    AF              ;save (AF) reg
05000 PLOOP   LD      A,(HL)          ;get a byte
05010         OR      A               ;test or 0H
05020         JR      Z,LAFIN         ;yes, jump
05030         CALL    OUTBYT          ;output byte
```

(Continued on next page)

```
05040           INC     HL              ;bump pointer
05050           JR      PLOOP           ;next byte
05060 LAFIN     POP     AF              ;restore (AF)
05070           RET                     ;return
05080 ;
05090 ;=== OUTBYT === Calls: $VDCHAR  (0033H)
05100 ;=====================================*
05110 ;   Output a character to the screen   *
05120 ;=====================================*
05130 ;
05140 OUTBYT    PUSH    DE              ;save (DE) reg
05150           PUSH    IY              ;save (IY) reg
05160           CALL    33H             ;put the byte
05170           POP     IY              ;restore (IY)
05180           POP     DE              ;restore (DE)
05190           RET                     ;return
05200 ;
05210 ;=== LPRINT === Calls: LPBYTE
05220 ;=====================================*
05230 ;  Output a line to the line printer  *
05240 ;=====================================*
05250 ;
05260 LPRINT    PUSH    AF              ;save (AF) reg
05270 LPNTX     LD      A,(HL)          ;get a byte
05280           OR      A               ;byte zero?
05290           JR      Z,LAFINX        ;yes, jump
05300           CALL    LPBYTE          ;print the byte
05310           CP      0DH             ;<CR>?
05320           JR      Z,LAFINX        ;yes, exit
05330           INC     HL              ;bump pointer
05340           JR      LPNTX           ;go again
05350 LAFINX    POP     AF              ;restore (AF)
05360           RET                     ;return
05370 ;
05380 ;=== LPBYTE === Calls: $PRCHAR (003BH)
05390 ;=====================================*
05400 ;  Output a byte to the line printer  *
05410 ;=====================================*
05420 ;
05430 LPBYTE    EXX                     ;save the regs
05440           LD      HL,37E8H        ;status byte addr
05450           LD      B,A             ;save character
05460 LP1       LD      A,(HL)          ;load status
05470           AND     0F0H            ;strip off status
05480           CP      30H             ;printer on?
05490           JR      NZ,LP1          ;no, get stuck
05500           LD      A,B             ;ret character
05510           CALL    003BH           ;send the byte
05520           EXX                     ;restore regs
05530           RET                     ;go back
05540 ;
05550 ;=== LPSTAT === Calls: nothing
05560 ;=====================================*
05570 ;  Check line printer to see if ready  *
05580 ;=====================================*
05590 ;
05600 LPSTAT    LD      A,(LPPORT)      ;get printer status byte
05610           AND     0F0H            ;strip off non-essentials
05620           CP      30H             ;is printer ready?
05630           RET     Z               ;yes, return to caller
```

```
05640           LD      HL,LPMSG        ;no, get message address
05650           CALL    PRINT           ;display message on screen
05660 STUCK     LD      A,(CTLKEY)      ;wait until <ENTER> pressed
05670           BIT     0,A             ;<ENTER> pressed?
05680           JR      Z,STUCK         ;no, wait some more
05690           JR      LPSTAT          ;yes, check status again
05700 LPMSG     DEFB    0DH
05710           DEFM    '>>==> Printer not ready, '
05720           DEFM    'press <ENTER> to continue.'
05730           DEFW    0DH
05740 ;
05750 ;=== INPUT (INPUTP) === Calls: INBYTE, OUTBYT, PRINT
05760 ;=====================================================*
05770 ;  Obtain line from the keyboard (with optional prompt)  *
05780 ;=====================================================*
05790 ;                                                       *
05800 ;         entry:  (HL) => input buffer                  *
05810 ;                 (B)  =  max. number of characters     *
05820 ;     (INPUTP):   (DE) => prompt message                *
05830 ;                                                       *
05840 ;         exit:   (HL) -  unchanged                     *
05850 ;                 (DE) => last character                *
05860 ;                 (C)  -  original contents of (B)      *
05870 ;                 (B)  -  actual # of characters        *
05880 ;                 (A)  -  termination character         *
05890 ;                                                       *
05900 ;         note:   The termination character is          *
05910 ;                 included in the character string.     *
05920 ;                                                       *
05930 ;=====================================================*
05940 ;
05950 QUESTM    DEFM    '? '            ;question mark prompt
05960           NOP                     ;prompt terminator
05970 INPUTP    EX      DE,HL           ;shift prompt addr to HL
05980           CALL    PRINT           ;display prompt message
05990           EX      DE,HL           ;shift buffer address to HL
06000 INPUT     EX      DE,HL           ;shift buffer address to DE
06010           LD      HL,QUESTM       ;point to question mark
06020           CALL    PRINT           ;display question mark
06030           LD      C,B             ;move maximum chars to C
06040           XOR     A               ;initialize the byte counter
06050           LD      B,A             ; contained in register B
06060           PUSH    DE              ;shift buffer address
06070           POP     HL              ; to the HL register
06080           DEC     HL              ;  and decrement it
06090 ILOOP1    CALL    INBYTE          ;get character from keyboard
06100           LD      (SAVEAZ),A      ; and store for later use
06110           CP      08H             ;is character a backspace?
06120           JR      NZ,STOREZ       ;no, jump to store character
06130           LD      A,B             ;get character count
06140           OR      A               ;any characters yet?
06150           JR      Z,ILOOP1        ;no, skip backspace
06160           LD      A,' '           ;yes, load a blank
06170           LD      (HL),A          ; and store in buffer
06180           DEC     HL              ;backup one character
06190           DEC     B               ; and decrement char counter
06200           LD      A,8H            ;load backspace character
06210           CALL    OUTBYT          ; and display on screen
06220           JR      ILOOP1          ;look for more input
06230 STOREZ    CP      0DH             ;is character <ENTER>?
06240           JR      Z,INRETZ        ;yes, go to return
06250           CP      1H              ;is character <BREAK>?
06260           JR      Z,INRETZ        ;yes, go to return
06270           LD      A,C             ;get maximum length
06280           CP      B               ;do we have max. chars?
```

(Continued on next page)

BASICFMT (Listing Continued, text begins on page 60)

```
06290           JR      Z,ILOOP1        ;yes, ignore input
06300           LD      A,(SAVEAZ)      ;no, retrieve character
06310           INC     B               ;increment byte counter
06320           INC     HL              ; and address pointer
06330           LD      (HL),A          ;  and store character
06340           CALL    OUTBYT          ;   and display character
06350           JR      ILOOP1          ;go to look for more
06360 SAVEAZ    NOP                     ;character storage area
06370 INRETZ    INC     HL              ;increment pointer
06380           LD      (HL),A          ; and store last character
06390           EX      DE,HL           ;restore registers
06400           CALL    OUTBYT          ;display last character
06410           RET                     ;return to caller
06420 ;
06430 ;=== INBYTE === Calls: KBSCAN
06440 ;================================*
06450 ;  Wait for a keyboard character  *
06460 ;================================*
06470 ;
06480 INBYTE    CALL    KBSCAN          ;look for a byte
06490           OR      A               ;zero returned?
06500           RET     NZ              ;no, return
06510           JR      INBYTE          ;yes, look again
06520 ;
06530 ;=== KBSCAN === Calls: $KBCHAR (002BH)
06540 ;================================*
06550 ;  Scan for a keyboard character  *
06560 ;================================*
06570 ;
06580 KBSCAN    PUSH    DE              ;save (DE) reg
06590           PUSH    IY              ;save (IY) reg
06600           CALL    2BH             ;scan keyboard
06610           POP     IY              ;restore (IY)
06620           POP     DE              ;restore (DE)
06630           RET                     ;return
06640 ;
06650 ;=== CPHLDE === Calls: nothing === equivalent to RST 18H
06660 ;=============================================*
06670 ;  Compare the contents of registers DE and HL  *
06680 ;=============================================*
06690 ;
06700 CPHLDE    LD      A,H             ;get MSB number 1
06710           SUB     D               ;compare MSB's
06720           RET     NZ              ;return if not equal
06730           LD      A,L             ;get LSB number 1
06740           SUB     E               ;compare LSB's
06750           RET                     ;return to caller
06760 ;
06770 ;    ***************************************************
06780 ;    *  Data storage areas, buffers, and control blocks  *
06790 ;    ***************************************************
06800 ;
06810 LINES     NOP             ;number of lines printed value
06820 QUOTE     NOP             ;quote processing logical variable
06830 REMARK    NOP             ;remark processing logical variable
06840 LL        DEFB    79      ;line length (less 6)
06850 PL        DEFB    60      ;page length (number of lines)
06860 PS        DEFB    6       ;page spacing (lines between pages)
06870 DCB       DEFS    32      ;File Control Block (FCB)
06880 BUFFER    DEFS    256     ;Input file buffer
06890 ENDBUF    EQU     $       ;end of input file buffer label
06900 OUTLYN    DEFS    512     ;secondary output buffer
06910 LINE      EQU     $       ;primary output buffer
06920 ENDPGM    EQU     $       ;end of program label
06930 CORE      EQU     ENDPGM-BASFMT   ;core used calculation
06940           END     BASFMT
```

End Listing

microSUB:MATH

Company: foehn consulting, P. O. Box 5123, Klamath Falls, OR 97601

Computer: Microsoft Fortran-80 and Cromemco Fortran IV (CP/M-80 only), Supersoft SSS Fortran (CP/M-80 and MSDOS), and IBM Fortran 77

Price: $250.00 ($25.00, manual only)

Reviewed by Elbert F. Hinson

microSUB:MATH is a rich collection of Fortran numerical-methods subroutines for use in the scientific and engineering fields. These routines are meant for serious Fortran programmers. The subroutines in this package provide many of the mainframe computer Fortran library routines for use on a microcomputer.

The package I received was for Microsoft Fortran-80 on a SSSD, 8-inch CP/M 2.2 disk. The routines are also available in almost any popular 5¼-inch disk format and are available for SuperSoft Fortran and Cromemco Fortran. By the time you read this article, a version should be available for the IBM PC.

The software is organized into six general areas. Two libraries, one for single precision and another for double precision, are provided for each area. The six areas are functions, interpolation, integration, matrix/linear systems, polynomials/nonlinear equations, and differential equations. Table 1 (at right) contains a complete description of the contents of each single-precision library. The double-precision libraries contain the same routines but provide the accuracy of double precision. The double-precision library names and subroutine names are prefixed with a D. For example, the library DFUNC is for double-precision functions and function DSINH is the double-precision hyperbolic sine.

Documentation

The manual supplied has a table of contents, an index, and tabs for each chapter. The introductory section gives an overall view of the available routines and how to use them. The method of linking the library routines with your loader is covered in sufficient detail for anyone familiar with Fortran. This is not a tutorial, so some knowledge of Fortran is expected if you wish to use these library routines.

Part of the introductory section is devoted to building custom libraries. This details which of the microSUB:MATH routines call other routines in the library and in what order the routines must be loaded in a custom library. Each of the six library sections has an individual write-up of each routine in the library.

The subroutine documentation is clear and easy to use. The manual provides the Fortran calling sequence, an explanation of the input and output parameters, an explanation of the method used and a sample program for each subroutine in the library. There are appendices for quick reference and test program results.

A set of test programs is provided on the distribution disk that exercises every subroutine in every library. Results of the test programs are provided in the manual for single precision but not double precision.

Performance

I tested a significant number of the subroutines but did not have time to test

Library	Subroutine	Purpose
FUNC (Functions)	SINH, COSH, TANH	Hyperbolic trigonometry
	ASINH, ACOSH, ATANH	
	CM, CD, CEXP, CLOG, CPWR	Complex math - multiply, divide, exponential, natural log, and raise to power
	CSIN, CCOS, CTAN	Complex trigonometry
	PR, RP	Polar to rectangular and rectangular to polar transform
	RJN, RJV	Bessel functions, integer and real orders
	ERF	Error function
	GAMMA	Gamma function
	SFRES, CFRES	Fresnel integrals, S(x) and C(x)
	SI, CI	Sine and cosine integrals
	CNST	Constants - PI, Euler's, and computer accuracy
INTP (Interpolation)	LIN	Linear interpolation
	LAGR	Lagrange interpolation
	UDVTB, UDVIN	Table for divided difference and solution for unequally spaced points
	EDVTB, EDVIN	Table for divided difference and solution for equally spaced points
	SPLTB, SPLIN	Second derivative table for cubic spline and cubic spline interpolation
	LAGDB	Double Lagrange interpolation
INTG (Integration)	SIMP	Simpson's
	ASMP	Adaptive Simpson's
	ROMB	Romberg
	GAUS1	Low order Gauss-Legendre
	GAUS2	High order Gauss-Legendre
MATX (Matrix/ Linear Systems)	RLU, SLU	LU decomposition and solution by Gauss elimination
	SOER, SOES	Gauss elimination with scaling and partial pivoting and solution from elimination
	GJOR	Gauss Jordan elimination
	MTM	Matrix multiplication

Table 1.

every one of them. I compared the results of most of the functions for random values against the CRC math tables. Function results appeared good for six digits of precision. Double precision should provide more accuracy if it is needed.

I ran parallel tests of many of the Matrix, Integration and Polynomial subroutines against the programs in the book *Fortran Programs for Scientists and Engineers* by Alan R. Miller. The results were the same for at least five digits in the single-precision versions. Although I did not specifically benchmark the results, the routines appeared to execute at about the same speed. I found it very easy to substitute the subroutines from the libraries for the subroutines in Mr. Miller's book. It took an awful lot of my time and typing to build the same thing that is contained in the library.

Support

I talked to Mr. Hugh Currin of foehn consulting about the package and some of my wish list. The results of the double-precision test programs should be included in later releases, but this present omission should provide no real problems to the user. I asked Mr. Currin if he had considered including the transcendental functions tangent, arcsin, and arccos that Microsoft left out of their library. He indicated that they could be included in a later release.

Conclusions

I highly recommend this package for any scientist or engineer who would like the power of mainframe subroutine packages on a microcomputer. There is a significant amount of scientific work that can be done on a dedicated microcomputer instead of fighting a busy mainframe. This package is very easy to use and contains a very powerful set of numerical subroutines for microcomputers.

(Continued from previous page)

Library	Subroutine	Purpose
	MINV	Matrix inversion
	JCBI	Jacobi, eigenvalues and -vectors of real symmetric matrix
	HQR1	HQRI eigenvalues of a real symmetric matrix
	INVIT	Eigenvector of a real symmetric matrix
	HQR2	HQRI eigenvalues of a general real matrix
	HQR3	HGRI eigenvalues and -vectors of general real matrix
POLY (Polynomials/ Nonlinear Equations)		Real root of an equation:
	HINT	Half interval method
	BRENT	Brent's method
	NEWT	Newton's method
	NEWTS	Nonlinear system of equations (Newton's method)
	JACM	Jacobian matrix (used in NEWTS)
		Orthogonal Polynomials:
	CHB1, CHB2	Chebyshev, first and second kind
	HERM	Hermite
	RJACB	Jacobi
	RLAGR	Generalized Laguerre
	RLEGN	Legendre
		Roots of Polynomials:
	PLYR1	Second, third, and fourth order
	PLYR2	One polynomial root (Muller's method)
	POLY	Polynomial evaluation
	PLYR3	Roots of polynomial (Muller's method)
DIFEQ (Differential Equations)	RGK	Runge-Kutta, fourth order
	ADMLT	Adams-Moulton, fourth order
	RKF	Runge-Kutta-Fehlberg variable step size

Table 1.

METAFILE Application Development System

Version 7.0 (Major_Updates. Minor_Updates)

Company: Sensor-Based Systems, 15 East Second Street, Chatfield, MN 55923-1297, (507) 867-4440

Computer: IBM PC or XT. IBM PCDOS version 1.1 or 2.0. Monochrome or color monitor. Minimum 128K RAM; 640K supported. One 2s/2d floppy disk drive plus one other storage unit (floppy or hard disk). Asynchronous communication adapter - RS232C. Printers supported: IBM Epson, NEC 3550, Dataproducts M200, Okidata 84

Price:$1050 (minor updates $25; major updates $350)

Reviewed by Morton F. Kaplon

The vendor's release describes METAFILE as follows:

"The METAFILE Software System creates a development and operational environment that includes integrated facilities for high-level programming, word processing, data base management, modeling, report generation, and communications."

This does indeed reflect the capabilities contained within METAFILE. The dictionary lists a variety of meanings for the prefix "meta," but clearly the one intended here is that of "beyond" or "transcending." And METAFILE certainly does extend the usual concepts associated with filing in its most general sense.

The potential users of this package could easily include a commercial enterprise requiring sophisticated management support, an applications programmer writing in a high-level language (METAFILE, in this case), and a stand-alone individual who likes to have a lot of capability at his or her fingertips. An example, which demonstrates both the program's integration and its power, is the ability to incorporate extensive numerical or logical calculations within a word processing environment and to store those calculations in either an evaluated or an unevaluated form so that the input can be changed, rather like a spreadsheet (though this is certainly not a spreadsheet), to obtain an output reflecting the change.

In order best to understand METAFILE's structure, we have characterized it as an operating system, called METAFILE ASSISTANT, that controls access to a series of integrated and interacting

programs. These programs include: Record Management; Text Management; Text Print Control; and METAFILE Command Language. These terms are from the manual. In more familar language the programs are, respectively: Relational Data Base; Full Screen Editor; Text Formatter; and METAFILE Interpreter.

METAFILE employs two distinct operational screens. One, which we will refer to as the COMMAND mode screen, can be considered as METAFILE's operating system screen, and the other, which we will call the EDIT mode screen, is that of the text editor.

COMMAND Mode

When you call METAFILE from DOS, the first screen presented prompts for a user ID (optional) and the date/time if not entered via a clock/calendar on boot to DOS; it then asks you to press Enter. The user ID is a METAFILE built-in function and could be easily adapted, using their programming language, to control operator file access. The next screen displays the COMMAND mode of METAFILE (Figure 1, below).

The screen is 80 columns wide by 24 rows deep. The failure to use the 25th row, available on the IBM PC, is probably related to the program's origins from the "Vector 3." These origins are reflected also in the way the function keys on the keyboard are ignored, using instead various Ctrl-letter combinations. These aspects show a weakness in design that Sensor-Based Systems states will be remedied in their next release.

The 23rd row is used for status information, and the 24th row is used for command area entry and for messages. The 22nd row is held blank to allow separation between the display area and the status lines, thus leaving a usable display area of 21 lines. Both the status area and that containing the METAFILE ASSISTANT are in reverse video.

The METAFILE ASSISTANT controls access to a set of integrated programs. METAFILE is essentially a menu-driven program, choices being made by positioning the cursor on an option and pressing the Return key; depending on the option selected, further menus may be presented for additional selection. Some commands can be entered directly by typing the appropriate command, thus bypassing some intermediate menu structure.

Perhaps the two most-used keys with METAFILE are Esc and ~, the latter not requiring the Shift key. ~ is referred to as the ATTN key, and when pressed it presents in the display area the relevant menu for the program in operation *or* the METAFILE ASSISTANT menu. The Esc key is implemented in the same way as in VisiCalc, Lotus 1-2-3, and many other programs: it takes you back one level in your calling sequence.

CALCULATOR

Of the five principal options, CALCULATOR offers the most immediate access. This option positions you directly in a screen insert, in reverse video and of the same contour as METAFILE ASSISTANT, from which you may perform a variety of numerical and logical calculations using the METAFILE Interpreter.

In the CALCULATOR mode, you can carry out any of the allowed METAFILE operations using numbers, strings, defined variables, and built-in functions for its operations. Calculations are carried out in floating decimal to 20 places. Included in the operations are add, subtract, multiply, divide, raise to power (**), operations on "dates," and the standard set of relational operations, for both numbers and strings, yielding a 1 if true or a 0 if false. The logical AND, OR conjunctions are included.

Built-in functions include natural logarithms, absolute value, random numbers, a variety of "date/time" functions, a function returning a one-byte binary equivalent of the value of its argument (this is used for printer control addressing), and a host of others related principally to data base structures. Arithmetic is performed strictly from left to right, and a generous use of ( ) is suggested to ensure that CALCULATOR does what is intended.

ACTIVE CONTEXT

The ACTIVE CONTEXT option offers a menu enabling selection from a listing of submenus. These summarize the active resources in METAFILE. They include: inter alia; the status of parameters established in the text editor, such as margins, search and translate values, page formats, printer status, etc.; structure of record files; listing of variables established with memory used; name of current drive; active files; and a host of other parameters defining the current status of the system. ACTIVE CONTEXT is presented in the same format as METAFILE ASSISTANT.

LOOKASIDE

The LOOKASIDE option is a particularly useful utility. Essentially, LOOKASIDE is a split-screen facility, but in METAFILE's realization, the alternate view is presented in the same reverse video setaside contour as the METAFILE ASSISTANT. Selecting LOOKASIDE presents you with a submenu of three choices: EDIT, UPDATE, and VIEW, as well as the option of typing in a command, as is offered in METAFILE ASSISTANT.

It is useful to think of LOOKASIDE as a second (albeit somewhat less enriched) command screen from which to exercise METAFILE's capabilities. Thus, selecting one of the options can lead to further menu choices, as from METAFILE ASSISTANT, or entering a command directly on LOOKASIDE's com-

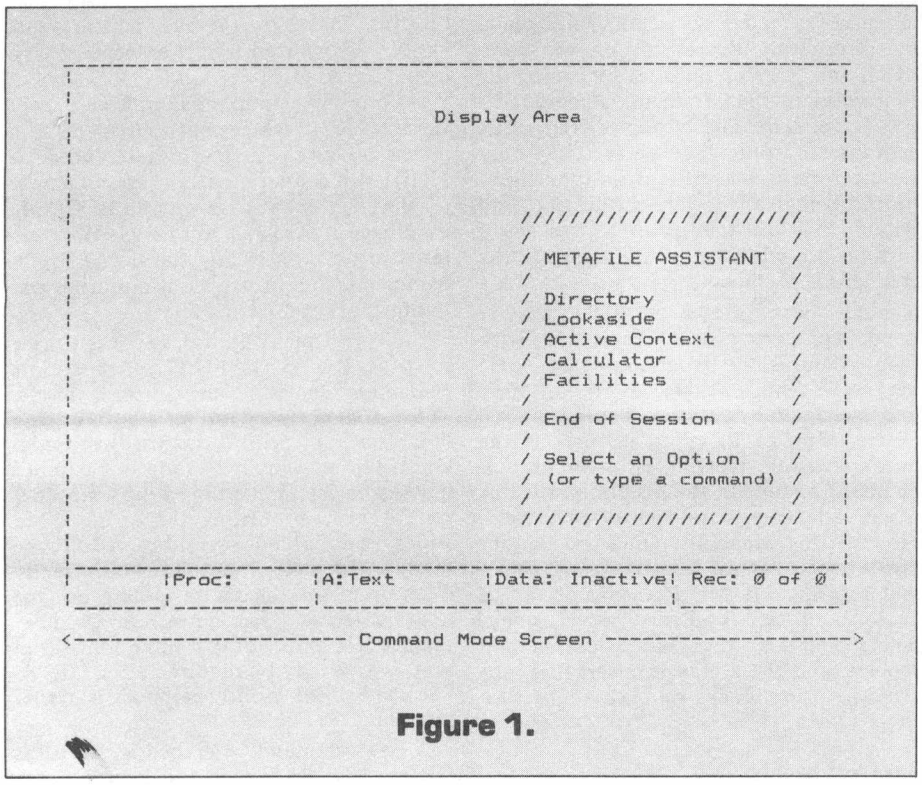


Figure 1.

mand line can enable you to move more directly to the task.

Selecting the EDIT option allows you to select a text for editing. Choosing the VIEW option allows a passive interaction with a file or record (like DOS's "type"). Choosing the UPDATE option allows operations to be performed with the data base. Other options may be entered directly. In LOOKASIDE, you have the same full power of the utilities available as if they had been called from the COMMAND screen.

Used in conjunction with the regular EDIT mode, this option allows the equivalent of "copying" from the LOOKASIDE file into the current text buffer by utilizing the include facility of METAFILE's editor. This allows you to look at what you may want to include in LOOKASIDE, edit it, write it out to file, and then read it back in, into the active text buffer.

DIRECTORY

The DIRECTORY option is concerned both with the management of the storage devices and with the active text buffer. Since METAFILE considers certain file types as reserved (e.g., those dealing with procedures, descriptions, and forms) and distinct from text files, this option offers the viewer the choice of a file listing by drive and type (and text is considered a type) or, under General Directory, by drive without restriction. Optionally, the current text buffer can be chosen. Once a file is selected, the user is presented the choices of EDIT, VIEW, or PRINT. Finally, data is presented on storage space availability and entry utilization. Some of the submenus of DIRECTORY are also available from other menus.

FACILITIES

The FACILITIES menu offers the largest number of submenus: nine, plus the usual choice of typing in a command. Each of these options in turn references further submenus. If METAFILE has an operational core, it is contained here. Included is access to basic operating system commands dealing with file management, such as storing, activating, erasing, renaming, transferring from one storage device to another, and establishing communications with other METAFILE systems via an "on line" (local network). Access to the editor is also available here.

Of particular importance is entry to the large resource of commands dealing with the data base; these include commands relating to its definition, maintenance, and interrogation, as well as the structures relating to form preparation and report generation. An additional option available relates to the procedures to "import" data base files from "foreign" formats into METAFILE.

EDIT Mode

The EDIT mode contains the (full) screen editor. The display area is 80 columns wide and 21 rows deep. Row 22 is blank, and rows 23 and 24 are used for status and prompts, respectively (row 25 is not utilized). The status line displays (in reverse video) the filename; line and column number of the cursor with respect to the upper left corner of the complete text; line and column number of the cursor with respect to the current screen (frame); margin status; mode status (wordwrap, insert or overtype, etc.); and percent of text space filled. Horizontal scrolling up to 250 columns occurs if wordwrap is not enabled at a column less than 80.

METAFILE's EDIT buffer is maintained entirely in memory and has a maximum capacity of about 63K. Due caution is required in movement between the EDIT and COMMAND modes so that the active EDIT file is not lost; METAFILE has inserted prompts to ensure that the buffer is not inadvertently lost.

The EDIT mode is used for a variety of purposes in addition to straight text editing. On one level it can be viewed as the editor for the METAFILE command language. In EDIT you can write a METAFILE procedure (program) and run it (note the similarity to BASIC). The editor is also used to create forms for data entry and to format reports.

Access to the various editing commands is via the ATTN (~) key where a menu is presented. Selection can be made by positioning the cursor on the command and pressing Enter or by typing the first letter of the command. A help facility is available. The editor is crisp in its response to commands. In fact, the only slow response was that of its type-ahead buffer, principally when scrolling beyond column 80. In that case, it was easily fillable, to which it responds with the standard IBM "beep."

As a word processor, EDIT is quite good, comparing quite favorably with many stand-alone programs. It has some very good features that many do not have and lacks some that would make it even better. One of its nicer features is the automatic reformatting (if wordwrap is enabled) of text in the insertion mode. A negative feature is the lack of any overall reformatting capability. Once the margins are set, you cannot return, change them, and reformat. It has a reasonable set of cursor and text controls: move, copy, delete line and text block, search and replace, and the ability to insert files into text, to print blocks of text, or to save them to disk.

Text Formatting

Extensive formatting commands allow headers, footers, centering, protection of text block on printing, multiple print copies, printing of selected pages, automatic page numbering and date inclusion, right justifying, and others. Practically any METAFILE command can be included in the print formatting, and printing can be done very easily from within METAFILE. The print control commands are included in the text (on separate lines) using *%command* at column 1, where *command* is any valid print command or any METAFILE command. Comments may be included by following the % symbol with a semicolon, followed by the comment.

Special Features

It is worth considering some of the unusual facilities of METAFILE. One of these is the ability to include operations within a text file. Any allowed METAFILE command or active variable can be accessed in the EDIT mode by placing the variable name or the METAFILE command within { }. The editor has a command named EVALUATE that displays the value of the variable or of the operation contained within { }. EVALUATE toggles between evaluated and unevaluated displays.

For printing or saving text to file, one can choose either the evaluated or the unevaluated mode. Since the variables defining record parameters are included here, it is clear that this offers an exceptionally rich tool in the generation of forms, reports, and letters, utilizing elements from the data base. It is this feature that makes mail merge and the printing of personalized letters extremely easy from METAFILE.

Consider a simple example: an array variable X(i), where each element is a name. A given name is referenced as {X(i)}. On calling EVALUATE, a name is displayed. For the case where the X(i) are numbers, if {X(2) * X(3) - X(4)} is entered in text, on calling the EVALUATE command, the substitutions are made and the formula evaluated.

As with many things, there is a price to pay. The METAFILE editor counts characters in the unevaluated mode. Thus, if {A} (containing on evaluation "Mr. Jones") is within the wordwrap margin in the unevaluated mode but extends beyond it on evaluation, wordwrap does not occur; on printing the string, "Mr. Jones" will extend beyond the wordwrap margin. However, the failure to correctly determine the character count on the unevaluated { } has more serious consequences for embedding printer controls in text, which may be required since METAFILE's current printer support is rather minimal.

The command to embed a control is {Code (#)}, where # is the decimal value

of a number in the print control sequence. Here the embedded format dominates in the determination of the column number for wordwrap, and it will always occur before the evaluated location. You are also offered the option of implementing printer control codes by text formatting; the format is: "%CONTROL *formula*" where *formula* is as above without the { }. This will not affect wordwrap, but it does not allow in text embedded controls.

It does not require too much imagination to see that it is possible to produce formatted spreadsheets using the { } utility. It is not a substitute for a true spreadsheet, since to change the value of a variable requires that the METAFILE "SET" command be used (equivalent to BASIC's LET VARX = xxx), but it does offer a very useful tool.

METAFILE's procedures for creating menus are particularly easy to implement. This is done by using [ ]'s to contain METAFILE commands or the names of executable METAFILE procedures as the options in the menu. An active text, with these choices enclosed within [ ]'s, is the basis for user-defined menus.

The menu and its options are formatted within the EDIT mode under a user-selectable filename with type MEN. A METAFILE procedure of the same filename but of type PRO is written (the manual lists the procedure; it is a mere seven short lines). The menu is called from a command line by typing in its filename. Under control of this program, the cursor keys control movement between options. Return selects an option, and when it is concluded the menu is presented again for further selection; pressing the Esc key concludes the menu session.

The options, as noted, can include METAFILE procedures so that the menu allows selection of user-written procedures to accomplish specified tasks. Since METAFILE allows a procedure to call others as subprocedures, it is clear that the language offers powerful capabilities for user definition and employment.

System Parameters

METAFILE has the implementation limitations listed in Table I (below) METAFILE admits three types of data elements: character (C), numeric (N), and data (D), with implementation restrictions noted in the table.

In addition to restrictions on implementation, another very important aspect of any data base is its operational performance in interrogating files. How long does it take to sort a file of 10,000 records on a single key? With qualifications? And so forth. Unfortunately, these questions are not addressed in the manual, and this reviewer has certainly not had the time to create such a base and test its performance. METAFILE, however, is not unique in failing to address these problems in its documentation.

It is not really meaningful to carry out a test on a small data base since it does not yield meaningful results for the potential user with a data base of significant proportions. This aspect poses a serious problem for any prospective user of any data base program, and it is not one that has been appropriately addressed. This is an area that calls for clarification for all such programs.

Installation

The package contains three Reference Guides, an Installation Guide, a "Configuration Device," and two double-sided, double-density disks. The program disk has 317,440 bytes of operational files, and the demo disk has 284,672 bytes of files. The program disk is so packed that if you are operating under DOS 1.1 you cannot get any DOS system files on it. Under DOS 2.0 you can just barely make a bootable disk with the system files, but when you go to enter your printer configuration (even as small a file as that), you obtain a "disk full" message.

Thus, for practical purposes a separate disk to boot the system is required, and you can then include such useful utilities as a clock calendar (whose input METAFILE automatically accepts); ProKey, which works quite well; and a RAM disk, which is quite useful. In exploring METAFILE, I used a boot disk formatted as indicated above, with an autoexec.bat file that, among other things, copied the program files in drive b: to the RAM disk (after a prompt to install in drive b: a disk with program files). I then put a data disk in drive b: (again on prompt) and was set to go.

METAFILE is copy protected via hardware. The disks per se are not protected, but to utilize them effectively you must connect the METAFILE "Configuration Device" to the COM 1 serial port. This device is a short cable having female and male connectors with a "black box" in between. The connector choices allow you to reconnect your modem (or any other device that may have been using the port) onto the cable and operate normally.

I tried my modem in that configuration and it worked without problems. Without the special cable connection, METAFILE can function but with only minimal memory (about 12 percent of system capability) and as such is useful only for demonstration. The device operates to restrict the system disk or its copies to a single machine for full utilization. There are the usual warranties to replace either the system disk or the "Configuration Device" within 90 days if either malfunctions.

Since METAFILE will operate with 128K RAM and the program files take up almost 320K, it is clear that many calls are going to be made to the program disk. If you do not have a hard disk, operating from a floppy can be rather slow; you could, however, use a RAM disk, which will afford some increase of speed even with a hard disk. METAFILE's instructions must be followed very closely for operation with a RAM disk. To operate with RAM, METAFILE is called from DOS with the command: metafile , memory:*xxx*K, where *xxx* is the amount of memory being reserved for METAFILE in units of 1000.

Since METAFILE's capabilities and capacities are affected by the amount of memory available (delineated for the user in a table in the Installation Guide),

1.	Maximum number of characters in a filename	8
2.	Maximum number of characters in a file type	3
3.	Maximum length of a record (characters)	1000
4.	Maximum number of records in a file	32000
5.	Maximum size of a text (approx. pages)	15
6.	Maximum length of a text line (characters)	250
7.	Maximum length of an item (characters)	235
8.	Maximum precision of a number (digits)	20
9.	Double quote (") not allowed in record item.	
10.	Backslash (\) not allowed in text line.	
11.	File and item names may contain only alpha numeric characters and underscores (_).	
12.	Date must lie between 01/01/1776 and 12/31/9999	

Table 1.

METAFILE Restrictions

reasonable care should be exercised in utilizing a RAM disk. Since at least 320K is needed for the RAM disk, a user contemplating that approach should, by and large, have a totally configured system with respect to memory. Of course if you operate from a hard disk, you have more options available.

In any event, METAFILE demonstrates some peculiarities with respect to RAM disks. I use two different programs. One utilizes the RAM disk driver contained in DOS 2.0, modified for higher capacity, and the other is Tall Trees' JFORMAT. METAFILE seems to accept the former with great tolerance, behaving well even when I forget to call it as indicated above. On the other hand, with JFORMAT, unless it is called in precisely the specified format, the system hangs up; if called correctly, it performs flawlessly.

There is one operational oddity, at least in the system I have, that is independent of the use of RAM disks or of any other peripheral hardware or software. After once calling METAFILE and closing a session to return to DOS, the DOS underlined blinking cursor is lost and does not return until the system is rebooted (warm does it OK); otherwise, DOS performs normally. I must admit that loss is a little disconcerting, but it is not disfunctional.

Documentation

The Installation Guide tells you how to use the demonstration facility (the demo disk). I find demos hard to evaluate. It is my belief the reasonably educated user is probably going to have more difficulty than a novice, principally because he or she just does not follow them with sufficient attention. On occasion I have asked my children or my wife to react to a demo disk of some program, and they have invariably been more satisfied with it than I was.

With that caveat, I was not overly impressed with the demo disk. It did introduce you to the various aspects of what the system can do, asked you to create some records, to enter some text, to correct it, to print it, to do some finds and replaces, to sort a file, etc. But I do not really believe that someone who does not have some understanding of what files, records, and fields mean can really use the demo and get very much out of it.

METAFILE is supported by three separate reference documents: spiral bound, 8.5 x 11 inches in size, with heavy paper covers.

Volumes 1 and 2 are labelled as User's Guides. Volume 1 is dedicated to A. Word/Text Processing and B. Record Management; Volume 2 to C. Introduction to Programming, D. Applications Design, and E. Glossary/Index. Volume 3, about twice the thickness of either 1 or 2, is labelled Reference Manual. These manuals have many faults. One of the most nettlesome is that they are "dirty." I found after using them even briefly that my work area was full of lint generated by the manuals and I was constantly vacuuming it.

The manuals are by far the weakest part of the package and do not, in fact, do it justice. Operationally, they are not organized for ready reference: there are no index tabs to allow quick selection of text sections, and while there are tables of contents for each of the major subject areas, the indexes are associated only with the glossary and with the Reference Manual. The latter two indexes are reasonably well presented, each organized alphabetically with a description and a page reference for each item, but there is a fair degree of redundancy as well as missing items. Included with the reference documents are a METAFILE User Reference Card (8.5 x 3.5-inch folder) and a Function Key Specification card of the same dimension.

It is very cumbersome, particularly when finding your way through a new and rather complicated package, to have to shuffle three separate documents, none particularly well designed for reference. The vendors do indicate that new documentation and a new format (the IBM notebook and slipcase style) is in preparation for future release. It is sorely needed.

The first two User's Guides are hands-on tutorials. Volume 1 takes you through a subset of the capabilities of the word processor and includes a tutorial on record management using a hypothetical personnel office as a case study. Volume 2 includes a tutorial on the programming facilities of METAFILE, again using a case study based on a set of employee records, and a final set of case studies dealing with the use of advanced commands and facilities, application guidelines, prototyping techniques, testing, debugging, and documentation.

The last three tutorials contain many useful examples and METAFILE programs that can be employed by the user, demonstrating such diverse aspects as mail merge, label addressing, form letter preparation, report documentation, and "converting" documents to METAFILE from foreign formats. The conversion presented is for record formats only, and I was unable to find any discussion dealing with the "importation" of text files. Some experimentation indicated that the METAFILE editor will accept only the purest of ASCII text files. Since a reformat capability is not included, the ability to import text files is relatively moot.

Displaying the hierarchy of a command structure, particularly with a program so richly structured as METAFILE, in a readily comprehensible fashion is important. This is an area where the documentation is also quite weak. Although the structure is presented, it is not done systematically nor in a way that gives an overall perspective; as a result, it is not useful for orientation or reference. The user is unable to discern the relationships between various units without an undue and uncalled for effort on his or her part. A tree diagram would have been most useful.

Printer Support

A few comments on printer support are in order. The documentation states that the IBM Epson is supported and it means just that: the bare-bones Epson without Graftrax. This is not made very clear, and certain printer commands indicated in the manuals are not realizable with any Epson other than the bare-bones IBM. It turns out that it is possible to embed printer controls in text, as well as in other formats, due to the enormous capabilities of the program, but it is almost impossible to discern from the manuals how to do this. I had to resort to the vendor for information.

Sensor-Based Systems is very responsive to requests for assistance and was able to head me in the right direction. As a result, I found it possible to utilize the full text print capabilities of the Epson MX 100 (and, I would judge, any other printer) since decimal sequences can be sent to the printer from within text files (with care, as noted above) or from formatting commands. Printer support is certainly one of the weakest operational aspects of the program.

Any serious user of METAFILE in substantive proportions will almost certainly need print spool capabilities. A print spooler is not included, and given the memory requirements of METAFILE, the most efficient way to obtain that would be with a stand-alone spooler.

Use of the Keyboard

Although a card indicates the METAFILE function key specifications, the word "function" is used operationally. The IBM function keypad is not used at all. In addition to a reasonably standard use of the cursor pad, 16 METAFILE-dedicated commands are effected by Ctrl-letter combinations. Included in the 16 are two cursor-control commands.

Some of the letters in the Ctrl-letter combinations have a relation to the command but many do not; it seems to me a significant improvement would be to assign to the function keys their related combinations. In fact, I reassigned them with ProKey so that they coincided, insofar as possible, with similar assignments I use on other programs (one of the great advantages of products like ProKey is cross-program standardization). When I

commented to the vendors on their failure to use the function keys, their response indicated that future releases would include a built-in macro facility that would allow the user that option, among others. This would be an enormous enhancement indeed.

METAFILE Programming Language

METAFILE's programming language encompasses a wide range of capabilities. It is primarily a nonprocedural language, although it contains a significant number of procedural commands. In the latter respect, it is structurally similar to BASIC, though the COMMAND names are different. In METAFILE, the command WRITE 5*45 will yield 225, just as in BASIC; PRINT 5*45 will do the same.

METAFILE procedures (programs) are identified by the file type (extension) PRO, just as BASIC programs are identified by the extension BAS, and as in interpreted BASIC, they can be run from the editor. All METAFILE commands must be entered in upper case, which given the IBM's lack of a CAPS LOCK indicator is somewhat of a nuisance.

An appreciation of METAFILE's capability can only be obtained by working with it, but it seems reasonable at least to list the classes of commands available and their functions (see Table II, page 83). Brief explanatory comments are noted in some cases.

Variables can be named and multidimensional subscripting is allowed as long as the total length of a data element is less than 235 characters. Variables are initialized by the command SET (with multiple initialization on one line allowed) and undefined with the command UNSET. Output is routed to SCREEN, PRINTER, or TEXT by use of the DEST command; default is SCREEN.

It seems useful to list the memory-dependent capacities of METAFILE as given in the Guide to Operations, since system performance is dependent on it. See Table III (page 83).

Summary

METAFILE is an excellent data base manager (record management, in METAFILE's lexicon) that utilizes a relational format. Each data record file has a data description specifying the attributes of the file and the items in its records. These are submitted in response to a command called DESCRIBE. Each item is capable of quite extensive description, including a comment line. Five files may be activated simultaneously.

Records may be updated, descriptions changed, new records inserted, and old ones deleted. Changes may be permanent or temporary, and the files may be searched and sorted with flexible querying commands. Fast lookup is implemented by a KEYING command. Columnar reports are produced with a LIST command and a row format with a ROW command. ITEMIZE will produce qualified records having non-null values.

The FORMAT commands allow extensive text management and preparation facilities, and flexible form and report generation is quite accessible. The ability to write METAFILE procedures to incorporate user-defined command sequences makes it possible to produce a combination of menus and procedures to permit system use by relatively inexperienced operators.

METAFILE goes well beyond data base management and word processing. The support from its programming language and the tight integration of the system offer the user enormous flexibility and suggest many interesting uses beyond those usually encompassed within those terms. If METAFILE gains acceptance in the user community, I would expect to see significant offerings of applications programs utilizing its facilities.

I am sure we have all heard the statement from high-level management: "I want to be able to press a button and see the status of XYZ." This is literally realizable with METAFILE. It is very easy to envisage writing a procedure with minimal prompts (whose implementation METAFILE makes easy) to load the appropriate file(s) and display the appropriate variables. Alternatively, the display could be done from the editor, exhibiting data variables within { } whose evaluation is accessed by hitting the Evaluation key. METAFILE allows scrolling (using the Tab key) through the variable sequence. The existence of a macro facility would enhance this even further.

LOOKASIDE makes a user's "calendar" readily available for viewing or for updating without leaving the current work session. The enterprising programmer will, I am reasonably sure, come up with additional applications. The only aspect really lacking is a graphics capability; the lack of this capability is certainly not an oversight since METAFILE is records and text oriented, not number oriented, though it does more than a creditable job in that area.

The integration achieved in METAFILE and the ease both of utilizing the facilities in the editor and of transferring data, accompanied by the capabilities of the ASSISTANT and the LOOKASIDE facility, certainly make this a strong competitor in the integrated program sweepstakes. All this without the need of a hard disk, a mouse, or an enormous memory capability.

METAFILE's manners are impeccable. The prompts are to the point and understandable. Its error handling facilities are clean and clear as to correction. As noted, the only hang up that occurred was when basic operational instructions were not faithfully followed. The only real quirk was that relating to the disappearance of the DOS cursor after leaving a session.

The more I have used METAFILE in the process of preparing this review, the more I have been impressed both by its capabilities and by its depth. Alas, I have also been more and more depressed by the poor documentation and the failure of the implementation to utilize the IBM PC screen and keyboard features and to respond to its weaknesses (here I am referring to the lack of any flags for CAPS LOCK or NUM LOCK status and the concomitant requirement of COMMAND entry in upper case).

By using the term "depth," I imply that the user will continually find the program opening up new opportunities for application and use. When I initially ascertained the price of the package and of the major updates, I was somewhat taken aback. After exploring its potential, and considering the price of other DBM systems currently in the market place, I am not as taken aback as I was. I would expect, however, that a program as costly as this would not be as lax in dealing with details of screen and keyboard usage nor exhibit the deficiency of printer support. The question of performance speed in the crucial area of data base application is just not answerable in the context of a single review.

This is not, in my opinion, a program for the novice, publicity claims notwithstanding. It may be true that the novice could do a few trivial things right off, so to speak, but to really utilize the capabilities of this program, users must take the time and effort to learn something about it. I believe they will be rewarded for that effort. To be able to effectively utilize the program's resources will require some programming skills, again claims to the contrary notwithstanding.

What do I conclude? If the new releases contain a significant improvement in documentation, more responsiveness to the IBM's screen and keyboard features (with due respect for strengths and weaknesses therein), and enhanced printer support, the METAFILE must be seriously considered by those requiring an effective data base and management support system.

■■J

PRINT CONTROL COMMANDS – 14

SESSION FACILITIES – 8

(start, stop, HELP, print, etc.)

PROCEDURAL CONTROL

Clauses	8 commands
Commands	9 commands

FORMULA FACILITIES

Calculations	6 commands
Conjunctions	2 commands
Formatting	7 commands
Character Manipulation	5 commands
Built-In Functions	29 commands

All of the above are executable from a METAFILE procedure (program) or, as appropriate, from a command line or within the editor.

TEXT PREPARATION FACILITIES – 30

(in the EDIT mode only)

FILE COMMANDS

Record Files	9 commands
Text Files	6 commands
All Files	4 commands (erase, rename, transfer, connect)

RECORD COMMANDS

Update	8 commands
Qualification	3 commands (criteria)
Navigation	7 commands (moving within files)

TEXT LINE COMMANDS

Update	4 commands
Navigation	8 commands (locations)

DATA ELEMENT COMMANDS – 3

FRAME COMMANDS – 9

(these relate to the generation of forms and reports, including user interaction)

REPORTING COMMANDS – 18

Table 2.

METAFILE Commands

Total Memory[1]	Text Area	Separate Text/Sort	Sort Area[2]	Variables[3]	Items[4]
126K	26K	no[5]	26K	2175	3859
192K	63K	no[5]	63K	2943	4352
256K	63K	yes	63K	2943	4352
>256K[6]	63K	yes	63K	4100	6400

[1] Total memory means that memory assigned to METAFILE and does not include any assigned to a RAM disk.

[2] ((KeyLength)+3) * 10 (#Records to be sorted) must be less than 10 times the sort area (in bytes).

[3] Total number of characters in variables' names, values, and control information must not exceed value in table.

[4] Within active data file descriptions, the length of all items' names plus 10 characters per name must not exceed number in the table.

[5] These systems share the same area for text and sort. If you edit a text then sort a data file, text just edited is cleared from memory and must be recalled for re-edit.

[6] >256K means 320K to 640K in steps of 64K.

Table 3.

METAFILE Memory-Dependent Capacities

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

More on MS-DOS EXEC Function

Morton F. Kaplon, of Pomona, NY, writes: "I read with some interest your article in the December 1983 issue of *DDJ* . . . . I certainly do agree with you that the documentation on the DOS function 4BH is less than transparent, though my prize for opacity in DOS is still easily won by Chapter 13, 'Using Extended Screen and Keyboard Control.'

"In any event, my approach was somewhat different than yours and the program I had created offers a somewhat more extensive demonstration of the utility of that call. It allows the user to enter from the program anything which can be entered from the command line, albeit in two steps if there is a tail to the command line.

"Given this program, I believe it is fairly easy to see how one can construct a 'DOS MANAGER' for the user who wants to be prompted through DOS." Morton was kind enough to provide his program on a diskette, and we are printing a slightly modified version of it in Listing One (page 88).

PC-DOS Manual Typo

PC-DOS 2.0's manual contains a small typographical error in Chapter 13 which may confuse programmers who are attempting to use the Extended Screen Control. The command for "Erase in Line" (usually known as Clear to End of Line) is "ESC [K", not "ESC [k" as given in the text.

IBM PC Junior

We have now received a PC Junior to play with and are making many interesting discoveries. The graphics support on the Junior is considerably enhanced over the standard IBM PC, with programmable palettes and the ability to display sixteen colors in medium-resolution graphics mode or four colors in high-resolution mode.

One issue that has been completely ignored in the popular magazines is the raw performance of the Junior. Apparently, everyone assumes that because it uses a 8088 microprocessor at 4.77 MHz, it will execute programs at a speed comparable to a normal PC. Well, friends, it just ain't so. just ain't so.

The PC Junior does not have dual ported memory for the video refresh buffer. Consequently, the video controller must "steal cycles" from the CPU while it is updating the screen. It turns out that two out of three memory cycles for the main machine RAM are allocated to the 6845 controller, and only one out of three for the 8088. Running several standard benchmarks, I found that the execution speed of a computation-bound program on the Junior is exactly 45% of its speed on a normal PC. Programs which perform a great deal of screen I/O appear to suffer even more, though I haven't measured this exactly yet.

The literature on the Junior leads one to believe that programs executing out of a ROM cartridge will execute much faster, since the video controller will not be accessing that memory. I'll check this out and provide more information next month.

The performance of PC-DOS 2.1, which is provided with the Junior, has also been degraded. Apparently the disk drives provided on the Junior have much longer seek and head settle times, so the BIOS disk driver's delay factors have been lengthened accordingly. Access to files by a disk-based program is painfully slow – just the process of opening a file seems to take over a second.

MS-DOS Volume Labels

I had surmised from reading the MS-DOS 2.0 manual that I could change the volume label on a disk by simply performing a search with an extended file-control-block for the first directory entry with a "volume" attribute byte, requesting a "delete" on any filename found in that manner, then requesting a "create" function with the new name. However, this path is fraught with peril. The delete appears to work OK and returns a successful status code, but it clobbers the file allocation table unmercifully (see Figures 1 and 2, page 87).

I was unable to figure out exactly what was going on here. Examining a dump of the FAT and directory sectors, it is readily seen that a volume label is simply a normal appearing directory entry with the attribute byte set to 8, date and time fields filled in, and the "starting cluster" and "file size" fields zeroed. Except for the attribute byte, this is just the same as a directory entry for a file that was created but never had any data written into it; however, a delete of such an empty file works fine. Evidently, finding the volume attribute bit turned on during a delete function sidetracks PC-DOS into some nutty logic or other.

By experimentation, I found that the following sequence will safely add or change a volume label under PC-DOS or MS-DOS 2.0:

1. Set the disk transfer address to a scratch buffer 128 bytes long.

2. Using an extended file control block with the format shown in Listing Two (page 92), perform a "search for first" function (#17). If the register AL is returned as 0FFH, then the disk has no label – go to step 5.

3. If function 17 returned register AL = zero, the buffer contains a simulated extended file control block with the volume name in bytes 8-19. Move your new volume name (11 characters) to buffer+24.

4. Passing the address of the scratch buffer in registers DS:DX, request a "rename" function (#23). If AL returns 00, the volume name was successfully modified and you are finished, otherwise something is horribly wrong with your system.

5. (Come here if the disk has no previous label.) Replace the wild-card characters shown in the extended file control block with the desired 11-character volume name. Passing the address of the extended fcb in DS:DX, request a PC-DOS "create" function (#22). AL returned as 0 implies success; AL returned as 0FFH means failure (usually due to a full directory).

I have provided an extract of the final working program as Listing Two (page 92) to get you started. Interestingly, this procedure *always* adds the volume label to the root directory no matter what your "current" subdirectory is.

I made another surprising discovery. Even though volume names were added in MS-DOS 2.0, the new version 2.0 extended file and record handling functions don't support adding or changing the volume labels. When I first tried this, I thought that I could probably use the new

CREAT call (3CH) and then use function 43H (CHMOD) to set the attribute on the new file to "volume"; however, the CHMOD call always returns an "access denied"*error code.*

Finding Size of TPA under MS-DOS 2.0

On the IBM PC, the Wang Professional Computer and presumably most other MS-DOS machines, there is a special BIOS call or some other method to get the amount of machine memory. Although it would be nice if RAM size could be obtained in a machine-independent way, there is no obvious MS-DOS function call that will do the trick. However, it can be done, although circuitously!

The method hinges on the "Modify Memory Allocation" call, MS-DOS function 4AH. It is defined as follows in the MS-DOS manual: "on entry, ES contains the segment of the block, BX contains the new requested block size in paragraphs. DOS will attempt to 'shrink' or 'grow' the specified block. If the call fails on a grow request, then on return BX contains the maximum block size possible . . . . "

Knowing that part of the MS-DOS procedure for loading an application program includes allocating a memory block to it, we can simply request an increase of our program's area to the maximum possible, inspect the amount of RAM that is actually given to us, and then add that to the segment address our program is based at in order to find the actual size of machine memory. See Listing Three (page 93) for an example subroutine that returns RAM size in Kbytes.

CP/M Discrepancies

While porting a program that performed ramdom file I/O from CP/M-86 to CP/M-68K, I discovered that they are not quite symmetric with respect to file control block structure. Specifically, bytes 31-34 (r0, r1, r2) have different meanings under the two operating systems (see Figure 3, at right). This was obviously done so that the lower 16 bits of the random record number would fall on a word boundary for the 68000, since 16-bit accesses on off addresses are verboten. However, it would have been nice if DRI had printed a warning about this in the manual somewhere.

■■J

(Listings begin on page 88)

```
Drive:B   Sector:1        Cluster:FAT#1   Dir:ROOT.DIR

          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

000000:  FC FF FF 03 40 00 05 60 00 07 80 00 09 A0 00 0B
000010:  C0 00 0D E0 00 0F 00 01 11 20 01 13 40 01 15 60
000020:  01 17 80 01 19 A0 01 1B C0 01 1D E0 01 1F 00 02
000030:  21 20 02 23 40 02 25 60 02 27 80 02 29 A0 02 2B
000040:  F0 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000050:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000060:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000070:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Drive:B   Sector:5        Cluster:        Dir:ROOT.DIR

          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

000000:  57 4F 52 4B 20 20 20 20 20 20 20 08 00 00 00 00
000010:  00 00 00 00 00 00 6F 14 32 08 00 00 00 00 00 00
000020:  46 4F 52 54 48 20 20 20 43 4F 4D 20 00 00 00 00
000030:  00 00 00 00 00 00 D9 08 6F 07 02 00 04 53 00 00
000040:  00 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6
000050:  F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6
000060:  00 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6
000070:  F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6 F6
```

Figure 1.

First block of File Allocation Table and Disk Directory before Delete performed on volume label using extended file control block. This disk contains only two directory entries and one file.

```
Drive:B   Sector:1        Cluster:FAT#1   Dir:ROOT.DIR

          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

000000:  FC FF FF 03 40 00 05 60 00 07 80 00 00 00 00 00
000010:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000020:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000030:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000040:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000050:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000060:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000070:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 2.

First block of File Allocation Table after Delete function call performed on volume label. Note that many of the clusters allocated to the normal file on the disk are now missing.

Byte name	Offset	CP/M-86	CP/M-68K
r0	33	1sb *	msb+
r1	34	2sb	2sb
r2	35	msb	1sb

*1sb = least significant byte
+msb = most significant byte

Figure 3.

Discrepancy in file control block structure between CP/M-86 and CP/M-68K.

16-Bit Toolbox (Text begins on page 86)

Listing One

```
        name    docom
        title   'DOCOM - Load and execute programs'
        page    55,132
;
; DO-COM.ASM --- Load and execute COM programs
; demonstrates use of PC-DOS function 4BH
; Morton F. Kaplan, 11 White Birch Drive, Pomona, NY 10970
;
; Definition of Function 4BH call:
; DS:DX points to ASCIIZ filespec
; ES:BX points to parameter block for load
; AL = function type:
;      0 is load and execute. Program must save registers
;        and must restore SS and SP.
;      3 is load only (may be used for overlays)
; Parameter Block
; AL= 0: Word segment address of environment string to be passed,0 value
;           for address will cause parent's environment to be inherited
;       DWORD pointer to command line to be placed at PSP+80h
;       DWORD pointer to default FCB to be passed at PSP+5ch
;       DWORD pointer to default FCB to be passed at PSP+6ch
;
; AL= 3: WORD segment address where file will be loaded
;      WORD relocation factor to be applied to the image.
;

; In this program, the filespec has to contain the filename and extension,
; a path may also be included.  Thus, to load and execute the program
; testmac.com on b: from drive a:, do as follows:
;
; A>docom
; Enter filespec to be loaded and run        (this prompt appears on screen)
; >b:testmac.com<enter>
; Enter command tail, if none then <enter>   (this prompt appears on screen)
; >
; Program then executes.

;
; Build this program from file DOCOM.ASM as follows:
; MASM DOCOM,DOCOM,DOCOM,DOCOM
; LINK DOCOM,DOCOM,,,
; EXE2BIN DOCOM.EXE DOCOM.COM
; ERASE DOCOM.EXE
;
;
; define necessary MACROs

dos     macro   fn                      ;fn is dos function to be called
        mov     ah,fn                   ;fn passed to ah
        int     21h                     ;dos interrupt
        endm                            ;end of macro

kb_in   macro                           ;chr typed at KB returned in al
        dos     1                       ;function 1
        endm
```

(Continued on next page)

Listing One

```
crlf    macro
        mov     dl,10                   ;line feed
        dos     2
        mov     dl,13                   ;carriage return
        dos     2
        endm

cue     macro                           ;perform CR-LF sequence
        crlf                            ;then display CUE mark
        mov     dl,'>'
        dos     2
        endm

bfr_kbin macro  bfr_addr                ;buffered keyboard input
        mov     dx,offset bfr_addr      ;dx points to input buffer
        dos     0ah                     ;dos function call
        endm                            ;end of macro

print_string macro bfr_locn             ;print string at bfr_locn
        mov     dx,offset bfr_locn      ;point dx
        dos     9                       ;function call 9 of dos
        endm                            ;end of macro

; END of Macro definitions

        codeseg segment para
        assume  cs:codeseg,ds:codeseg,es:codeseg

        org     100h                    ;required for COM

start:  jmp     docom                   ;jump over data section


error1    db    'Memory Shrink Failure','$'
error2    db    'Error in Loading Program','$'
prompt_0  db    'Enter filespec to be loaded and run ','$'
prompt_1  db    'Enter command tail, if none then <enter> ','$'

stk_ptr   dw    0                       ;save contents of SP
stk_seg   dw    0                       ;save contents of SS

                                        ;PARAMETER BLOCK
                                        ;
parm_blk dw     0                       ;segment address of environment
                                        ;
                                        ;string passed,use parent
                                        ;pointer to command line at
                dw      offset psp80+1  ; at psp+80h, +1 needed
                dw      ?               ;to allow for structure of
                                        ;psp80 using buffered
                                        ;input from keyboard
```

```
                                        ;
                dw      offset fcb1     ;pointer to default FCB1 at psp+5ch
                dw      ?
                dw      offset fcb2     ;pointer to default FCB2 at psp+6ch
                dw      ?

                                        ;DATA FOR PARAMETER BLOCK
                                        ;
                                        ;buffer of 20 char for buffered
                                        ;line input, can be increased
filespec        db      20,?,20 dup (?)
psp80           db      32,?,32 dup (?) ;32 chars in buffer,increasable
fcb1            db      1               ;drive id
                db      11 dup ('?')    ;filename and extension
                db      25 dup (0)      ;rest of fcb
fcb2            db      1               ;drive id
                db      11 dup ('?')    ;filename and extension
                db      25 dup(0)       ;rest of fcb

docom:                                  ;start of effective code
                push    cs              ;initialize DS and ES
                pop     ds
                push    cs
                pop     es              ;initialization finished
                mov     bx,50           ;save 50*16 bytes for program
                dos     04ah            ;es already set to code segment
                crlf
                jc      not_set         ;check for memory shrink
                print_string prompt_0   ;display prompt
                cue
                jmp     over_er1

not_set:                                ;display error message
                print_string error1
                mov     dx,ax           ;print error code in ax
                add     dl,48           ;convert to ascii
                dos     2               ;display on kb
                jmp     quit

over_er1:
                bfr_kbin filespec       ;get filespec
                crlf
                sub     ax,ax           ;clear to 0
                mov     al,filespec+1   ;get length of string input
                mov     di,offset filespec+2
                add     di,ax           ;to end of string
                mov     byte ptr [di],0 ;terminate with 0

                print_string  prompt_1  ;prompt for command tail
                cue
                bfr_kbin psp80          ;get command tail
                crlf
                sub     ax,ax           ;clear to 0
```

(Continued on next page)

Listing One

```
                mov     al,psp80+1      ;get length of string input
                mov     di,offset psp80+2
                add     di,ax           ;to end of string
                                        ;terminate with carriage return
                                        ;and null byte
                mov     byte ptr [di],13
                mov     byte ptr [di+1],0
                mov     cs:stk_ptr,sp   ;save SS and SP in
                mov     cs:stk_seg,ss   ;variables addressable by CS
                mov     di,offset parm_blk
                mov     [di+4],es       ;seg address to psp80
                mov     [di+8],es       ;seg address to fcb1
                mov     [di+12],es      ;seg address to fcb2
                                        ;point to parameter block
                mov     bx,offset parm_blk
                                        ;point to ASCIIZ filename
                mov     dx,offset filespec+2
                mov     al,0            ;load and execute
                dos     4bh             ;dos function to execute program
                mov     ss,cs:stk_seg   ;restore SS:SP pair
                mov     sp,cs:stk_ptr
                jnc     quit            ;jump if no load error
                print_string error2     ;print error message
quit:           dos     4ch             ;back to dos,
                                        ;easiest way to terminate, can also
                                        ;pass code back with this call
codeseg         ends
                end     start
```

End Listing One

Listing Two

```
                .
                .
                .
                mov  dx,offset buffer
                mov  ah,26          ;set disk transfer address
                int  21h            ;to scratch area (DS:DX) for
                                    ;dictionary search.
                mov  dx,offset xfcb ;DS:DX=addr extended fcb
                mov  ah,17          ;"search for first match"
                int  21h
                cmp  al,0ffh        ;successful?
                je   no_label       ;no label on disk, jump.
                jmp  label_found    ;label found, jump.
                .
                .
                .
xfcb            db   0ffh           ;flag signifying extended fcb
                db   5 dup (0)      ;reserved (should be zeroes)
                db   8              ;volume attribute byte
                db   0              ;drive code (set by program)
                db   11 dup ('?')   ;wild card filename & ext.
                db   25 dup (0)     ;remainder of fcb (not used)

buffer          db   128 dup (?)    ;buffer for directory search
```

End Listing Two

Listing Three

```
;
; Return memory size (Kbytes) in AX.
; Calling arguments:  none.
: Destroys AX, BX, CX.
; This routine assumes that ES is pointing
; to the Program Segment Prefix (true
; on original entry from MSDOS for either
; an EXE or a COM type program)
;
ramsize   proc      near
          mov       bx,-1       ;request 65535 paragraphs!
          mov       ah,4ah      ;MSDOS modify block function
          int       21h         ;returns paragraphs available
                                ;in register BX
          mov       ax,cs       ;add that to the segment of
          add       ax,bx       ;the base of our program
          mov       cx,6        ;then shift paragraphs right
          shr       ax,cx       ;six places to get Kbytes
          ret                   ;back to caller
ramsize   endp
;
```

End Listing

(Continued from page 27)

References

[1] *EXPERT-2 - A Knowledge-Based Inference Program.* Available as a document or document with source code on disk from Mountain View Press, Inc. (P. O. Box 4656, Mt. View, CA 94040), Miller Microcomputer Services (61 Lake Shore Rd., Natick, MA 01760), or Helion, Inc., Box 445, Brownsville, CA 95919.

[2] Richard Duda and John Gaschnig. "Knowledge-Based Expert Systems Come of Age." *Byte*. September 1981.

[3] P. H. Winston and B. K. P. Horn. *LISP*. Addison-Wesley.

[4] George W. Miller. "Weather Forecaster." *COMPUTE!* August 1983. See also Tom Fox, *Unique Electronic Weather Projects*. Howard Sams & Co., 1978.

[5] *Weather Training Manual*. Benton Harbor, MI: Heath Company.

(Continued from page 40)

I invite you to improve any part of what I have presented.

References

[1] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Trans Info Theory*, Vol. IT-22, No. 6, pp. 644-654.

[2] M. E. Hellman, "The Mathematics of Public Key Cryptography," *Sci Amer*, February 1979, Vol. 241, No. 2, pp. 146-157.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures & Public Key Cryptosystems," *Comm ACM*, February 1978, Vol. 21, No. 2, pp. 120-126.

[4] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd ed, Reading, MA: Addison-Wesley, pp. 250-268, 374-380, 386-389, 442-443.

[5] H. T. Gordon, "Fast Divisibility Algorithms," *DDJ*, June 1983, No. 80, pp. 14-16.

[6] S. G. Akl, " Digital Signatures: A Tutorial Survey," *Computer*, February 1983, Vol. 16, No. 2, pp. 15-24.

[7] D. W. Davies, "Applying the RSA Digital Signature to Electronic Mail," *Computer*, February 1983, Vol. 16, No. 2, pp. 55-62.

[8] R. DeMillo and M. Merritt, "Protocols for Data Security," *Computer*, February 1983, Vol. 16, No. 2, pp. 39-50.

[9] D. E. Denning, "Protecting Public Keys & Signature Keys," *Computer*, February 1983, Vol. 16, No. 2, pp. 27-35.

Additional References

[1] B. Schanning, "Securing Data Inexpensively Via Public Keys," *Computer Design*, April 5, 1983, pp. 105-108.

[2] J. Smith, "Public Key Cryptography," *Byte*, January 1983, pp. 198-218.

C/UNIX PROGRAMMER'S NOTEBOOK

by Anthony Skjellum

This month's Notebook includes reader response to my December 1983 column, where I discussed perceived deficiencies in the Unix operating system, as well as a follow-up discussion on runtime libraries and link format incompatibilities.

It was also my intention to discuss two C compilers that support the large memory concept on the 8086. But since I have yet to receive one of the compilers, this comparative review will have to be included in a future column.

Several more letters have come in concerning C layout standards. While we don't have room to include their comments at present, they will undoubtedly lead to several columns on related topics of interest.

Comments about Unix

In the December column, I mentioned several aspects of Unix that I believe constitute weaknesses. Tony LiCausi of Canoga Park, California, sent me a letter with his dissenting opinions. He begins by remarking:

"Unix was designed for the programming environment, for use by folk already familiar with computers. Unnecessary I/O slows down a program's performance and hence should be eliminated. Thankfully, verbosity is often eliminated in Unix as the default option. However, many of the programs do have command line switches to increase the progress reporting of the program."

Unix was designed to circumvent the flaws of existing operating systems. Under Unix, modular programs are coupled via pipelines to create a convenient working environment and to reduce repetition of effort. Furthermore, Unix offers unequalled portability between large and small computer systems. Since it is successful in these respects, Unix's popularity is growing; hence, more and more computer users are gaining access to machines that use Unix. Many of these users are familiar with computers and programming, while others are novices to both computing and Unix.

Let's assume that a user is well versed in programming but is new to Unix; according to Mr. LiCausi, this user belongs to the group of people for whom Unix was designed. I'll assume that the user has an application in mind, since the average user does not write programs as ends unto themselves. To succeed in implementing an application, the user must learn about various aspects of Unix, including various shell commands, system calls, and runtime libraries.

I submit that this type of user will be slowed down significantly by the overall poor quality of the Unix documentation. Since examples are typically both terse and complicated, the typical user could spend an enormous amount of time circumventing a single snag. Imagine linking the mathematical function library as part of a C compilation. Two seemingly possible ways for doing this are:

```
cc -o test test.c -lm
        or
cc -lm -o test test.c
```

It turns out that the former method is correct, while the latter produces undefined function error messages at the linker stage. However, the documentation does not indicate that the "-lm" must be near the end of the line.

Our hypothetical user will also be slowed down by the cryptic nature of the error messages. True, the more sophisticated users require little or no prompting, and only gentle nudges when they make an incorrect entry, but the vast majority of users probably don't belong to this category.

Finally, the user will be slowed down by inexplicable and undocumented bugs in systems such as **troff** and **eqn**, which I mentioned before. Furthermore, circumventing bugs requires highly specific knowledge, and the solutions, even if available, may not be widely known. On the Unix machine I use, only a small group of people is aware of the procedure for avoiding some common but annoying bugs in **eqn**.

Mr. LiCausi makes a very valid point concerning this:

"Software bugs are not acceptable in any form at any time. Unix from Bell Labs is sold as unsupported software. Their attitude is, 'OK, we'll sell it to you, but we don't want to hear from you.' There is no mechanism to report or correct bugs at the source. Source code is not typically available to the average user. Source code is guarded on Unix like gold, just like any other commercial software. Having done debugging of the f77 compiler on one system, I found access to other software (for examples and integration) severely limited."

Thus, while Mr. LiCausi contends that Unix was designed for people with programming experience, he sets his standard for "unnecessary I/O" and "program performance" based on the most sophisticated level of Unix user. Furthermore, it is still not clear to me why the sophisticated users should be more willing to stand for poor quality documentation and user-unfriendly software than everyday users. Finally, it is not at all apparent to me from practical experience that "many" of the standard shell commands support extended progress reporting.

Mr. LiCausi goes on to state:

"The secret society of Unix is partially the result of the paranoia of 'unauthorized access.' The remainder is due to the lack of commercial incentive. The incentive has obviously increased, as can be evidenced by the proliferation of books and articles on Unix and C."

I stated in my previous column that some users perceive Unix as a secret society because they cannot grasp certain aspects of its operation based on the written and on-line documentation alone. Only through initiated members can the knowledge (and undocumented tricks, patches, etc.) be obtained. However, other popular operating systems do not have the same aura as Unix. For example, CP/M users are not considered a "secret society" even though CP/M has nontrivial aspects. Nor do minicomputer operating systems such as VAX/VMS (DEC) and AOS/VS (Data General) have the mysterious quality of Unix. That's because they have much more documentation to explain the nitty-gritty of using the system.

Mr. LiCausi finishes by stating:

"Unix is not user-unfriendly. Admittedly it is beginner-hostile. Changes will be required to adopt Unix en masse (i.e., for the micro), but I hope that it does not spell the end of 'silent' software."

My final counter-remark is that most users never become sufficiently sophisticated in enough diverse aspects of Unix to escape the feeling of user-hostility. I personally still run into difficulties, even though I've used Unix for almost four years. As soon as I use aspects of Unix that are not part of my daily routine, "gottchas" appear.

Other Points of View

H. T. Gordon of Berkeley, California, sent in a letter concerning Unix. He writes:

"It was a pleasure to read the Skjellum critique of [Unix] in *DDJ* No. 86. Since C/Unix is becoming the sacred cow of the computer Brahmins, few have the courage to point out its flaws. It was designed by one of the elite, for elite users. To them it's concise and elegant, even 'obvious.' Only rarely is any explanation needed. If it is, a terse one will do. Prophets inspired by the high gods cannot waste their time spelling things out for mere mortals in words of one syllable."

I don't think that the state of C/Unix is as bad as Mr. Gordon suggests. While there are problems, I think that Unix has many strengths and these strengths will be the basis for future operating systems. It is the purpose of this column to examine strengths and weaknesses of C and Unix, and to propose solutions as well as to point out the problems. I encourage readers to pose solutions to these and other problems they encounter.

Low Level Input-Output in C

Patrick Cawood of Los Angeles writes:

"I read with great interest your last article on tendencies in Unix to produce poor operator interaction programs. I seem to have met some of the same in C.

"In order to provide a secure operator interface, one does not echo a keyboard character to the screen until it has been examined and approved by the program. But **getch()** function automatically echoes – even line feeds, up arrows, etc.! Function **putch()** provides an automatic line feed after printing on the screen!"

He goes on to state:

"I simply cannot believe that anyone would wittingly design these functions as they are, or not provide any alternative hardware interfaces. Especially considering some of the tasks I've heard were written in C. But perhaps these people writing serious software were all forced to write their own hardware interfaces."

The problem that Mr. Cawood is referring to exists in a number of non-Unix C compilers. To begin with, let's review the problem in the Unix environment. Under Unix, **putc()** and **getc()** acquire and return a single character, respectively. To offload the host system, however, many terminal interface boards programmatically handle the user input-output in lines. Thus, before any input is received by the host, a whole line must be entered. Naturally, the characters are echoed by the terminal interface hardware/firmware, and only limited line editing is permitted. On output, a whole line is buffered up before transmission to the terminal.

This process can be overcome by use of the "raw" terminal mode (raw implies no host character processing). In this mode, the program is completely responsible for input-output. This mode is much more expensive in terms of input-output cost, since the host must handle an interrupt for each input character and perform an output request for each printed character. However, as I mentioned in the previous column, this is the only way for a program to get full control of what is entered and appears on the display device.

With the introduction of C to microcomputers, compiler implementors often based the behavior of the runtime libraries on their Unix experiences rather than on *The C Programming Language*. Whole lines are prebuffered by typical C runtime libraries before a single character is received by **getc()**; conversely, a whole line of output is internally buffered before it is printed on the display. Thus, the runtime libraries of microcomputer C compilers often emulate the terminal hardware found on real Unix systems.

However, this is not what **putc()** and **getc()** are supposed to do. In the truest sense, raw mode is the fundamental terminal mode. These functions should really work on a character-by-character basis. C libraries should permit selection between the "raw" and "cooked" modes (and echo and no-echo modes), thus permitting input-output flexibility without resorting to assembly language routines.

One C compiler that correctly handles low level input-output is the Q/C compiler from The Code Works. Single characters, for example, are required only when a **getchar()** call is made. While it doesn't seem possible to turn off the echo, this can be effected under Q/C since The Code Works is kind enough to provide runtime and compiler source code.

Link Formats

In the October 1983 column I discussed incompatibility between link formats. Readers had quite a bit to say about this and their comments are summarized here.

Guy Scharf of Mountain View, California, writes:

"I find the incompatibility of linkage editor formats to be a real problem. For example, I want to use Digital Research's Access Manager and Display Manager 86 with C. I would prefer CI-86 (because I am used to it) but have to switch to DRI's C because of the link format. Another compiler to learn and idiosyncrasies to surmount."

He concludes: "I'm not sure what to do about this problem (except complain)."

David D. Clark of State College, Pennsylvania, writes:

"The big problem isn't really the format of linkable files. Even compilers that use Microsoft's M80 and L80 will not allow linkage to code produced by different compilers. The function-calling protocols vary tremendously from compiler to compiler. BDS C and Q/C have fairly straightforward function-calling protocols. Eco-C, on the other hand, has a tortuous function-calling sequence. And even though Q/C and Eco-C use the Microsoft assembler and linker, the code files produced by them are not compatible."

The problem that Mr. Clark mentions is also present in the 8086 (MS-DOS, CP/M-86) world. Code from different compilers cannot be mixed because:

1. Each requires its own main function.
2. A wide variety of link formats exist.
3. Calling conventions differ between compilers.

The first two points are essentially insurmountable problems from the end user's point of view. However, the third point can be overcome by adding dummy routines to convert calling conventions.

Mr. Clark makes some additional points concerning the deficiencies of the 8080 Microsoft .REL (relocatable) format. These are of interest since many 8080 C compilers rely on this format. His comments are summarized in Table I (below).

It is obvious that an enhanced standard is necessary for the CP/M-80 world. Even under MS-DOS, where Microsoft has enhanced the linkage editor format, problems still exist.

Table 1.

Deficiencies in the Microsoft 8080 .REL Format and Related Software

1. M80 and DRI's RMAC assemblers support only six unique characters (all upper case). This is awkward for many purposes.
2. While the .REL format apparently will handle seven unique characters, neither M80 nor RMAC supports this.
3. Apparently the .REL format, M80, and L80 were designed to work with the Fortran-80 compiler, which permits only six character symbols. This is an old standard and does not reflect the needs of today's compilers. The absence of case sensitivity in symbols is especially limiting.

(Continued on page 100)

by Michael Wiesenberg

TEXnical Information

If Stanford's Donald Knuth has his way, typesetting will no longer be an esoteric trade practiced only by a privileged and knowledgeable few. Computerized typesetting with its almost infinite flexibility permits individuals to set their own copy. No longer need the wasteful method be followed of one person producing copy on a word processor and printing that copy out on a daisy-wheel printer for another individual to reenter on the front end of a phototypesetter. That, of course, introduced new mistakes by the person doing the typesetting, an individual often untrained in the discipline addressed by the original author and thus unlikely to recognize many typographical errors.

Also the typesetting equipment, even if computerized, does not have error-checking facilities like the spelling and grammar checkers available on microcomputers. Sometimes the only interface between the supposedly computerized phototypesetting front end and the phototypesetter itself is not even a direct link but an antiquated paper tape. A completely new tape has to be punched each time a particular document is updated.

Some people predicted the death of the typesetting industry, or at least danger to members of the typesetting unions, when Knuth released **TEX** (pronounced *tek*) in 1978; but in fact what will happen is that typesetting will become more universally accessible, creating more jobs. When word processing was introduced, it did not – as some doomsayers had predicted – render secretaries obsolete; it just moved some of them from the typewriter to the terminal. What *did* happen was that more documents were produced. With the proliferation of computerized typesetting, we'll just see more good-looking documents.

Now, how can you get TEX onto your TRS-80 or Apple? Well, that's the bad news. You can't. Until recently, TEX was available only for mainframes. But here's the good news. You can get it on a system that has, depending on configuration, a minimum of 500K of main memory. Such systems range from the IBM PC to Hewlett-Packard computers. Since the TEX software is in the public domain, you could theoretically put it onto a virtual memory system with as little as 128K memory, but you'd have to adapt the software yourself and write the drivers for whatever output device you needed. That shouldn't be too hard for you Pascal wizards, right?

If you plan to implement TEX yourself and want the public domain software, you can get it from Maria Code by special arrangement with the Computer Science Department of Stanford University. 1200-foot tapes in various formats are $82 per tape if you supply the tape, or add $10 if you don't. The generic tape contains TEX82, WEB, fonts, and other miscellany; you need a Pascal compiler.

The TEXbook

Knuth's first version, TEX78, was written in SAIL. TEX82 is written in WEB, a system of structured documentation that combines the features of both a programming language (Pascal) and a document formatter (TEX). To illustrate Dr. Knuth's marvelous sense of humor (also found, by the

```
If the generator is now tuned to $f_0$, where the receiver power gain
is $G(f_0)$, the total power output will be
$$P=FkT\int^\infty_0 G(f)\, df+SG(f_0)$$
where $S$ is the available signal power. If $S$ is now adjusted so that
$P=2N$, i.e., so that the reading of the square law detector doubles, then
$$FkT\int^\infty_0 G(f)\, df=SG(f_0)$$
$$F={S\over kT{1\over G(f_0)}\int^\infty_0 G(f)\, df}={S\over kTB}
  \eqno{(3)}$$
where
$$B={1\over G(f_0)}\int^\infty_0 G(f)\, df=\hbox{effective noise bandwidth}$$
In order to find $B$ it is usually necessary to measure the power gain
as a function of frequency and integrate the curve graphically. This
method has major disadvantages, however, since it is time-consuming
and since $G(f)$ may vary with different conditions of adjustment.
A further disadvantage is the difficulty of determining $S$ with the required
accuracy at the low levels involved.
```

If the generator is now tuned to f_0, where the receiver power gain is $G(f_0)$, the total power output will be

$$P = FkT\int_0^\infty G(f)\,df + SG(f_0)$$

where S is the available signal power. If S is now adjusted so that $P = 2N$, i.e., so that the reading of the square law detector doubles, then

$$FkT\int_0^\infty G(f)\,df = SG(f_0)$$

$$F = \frac{S}{kT\frac{1}{G(f_0)}\int_0^\infty G(f)\,df} = \frac{S}{kTB} \qquad (3)$$

where

$$B = \frac{1}{G(f_0)}\int_0^\infty G(f)\,df = \text{effective noise bandwidth}$$

In order to find B it is usually necessary to measure the power gain as a function of frequency and integrate the curve graphically. This method has major disadvantages, however, since it is time-consuming and since $G(f)$ may vary with different conditions of adjustment. A further disadvantage is the difficulty of determining S with the required accuracy at the low levels involved.

way, in his *TEXbook*): you pass a .WEB source file through the TANGLE program to produce a .PAS Pascal file or through the WEAVE program to produce a .TEX file (that can now be processed by TEX). ("Oh what a tangled web we weave. . . .")

You cannot use TEX without a copy of *The TEXbook*. It is a delight to read, though really meant only for actual users of the software because it has a lot of exercises for you to try out. If you don't do them, you won't be able to follow what he's talking about. If you are familiar with his monumental series, *The Art of Computer Programming* (considered by most programmers to be the best thing written about programming), you have an idea of the level of expertise required for *The TEXbook*. Published jointly by Addison-Wesley and the American Mathematical Society, the book may be ordered from the latter for $15 (including p. and h., or add $2 for first class, $3 for airmail outside the U.S.). For 438 pages, most of it in nine- and ten-point type, that's a bargain!

HPTEX

The version I have been using is HPTEX, Hewlett-Packard's implementation of TEX82, on a 68000-based HP series 200 Pascal workstation, model 36. The software costs $4000. A typical configuration includes 9836A (the workstation with 500K RAM, Pascal, BASIC, UCSD editor, and several other utilities), $12,110, three more 256K memory boards at $1060 each, a disk interface for $605, DMA card for $500, 65Mb hard disk, $17,350, and the HP2688A desktop laser printer for $26,000. (You could save some money by substituting a model 26: you need 1.25Mb main memory, not necessarily the 1.75 I've been using; you can "get by" with 15Mb storage – although 65 is recommended – and you don't necessarily need the high-speed memory access, which also eliminates the need for the disk interface card.)

With this system, I can literally typeset an entire book by myself. I can choose from any of over 340 fonts, which come in sizes ranging from five to 40 points. A powerful macro set allows easy specification of any format. Dr. Knuth originally developed TEX so that complicated mathematical formulas in his books would be set exactly as he wanted, rather than as the typesetter felt it should look. TEX really shines at this, but it also excels at easily setting tables and other complicated formats.

HPTEX comes with an extensive set of macros that permit, among other things, multicolumn mode, with each column using a different font and a different measurement between lines, automatic generation of table of contents, and automatic indexing. I can define macros myself that permit juggling figures and tables around in a manual without worrying about changing the numbers (what happens to Figure 3-1, for example, and all the figures thereafter, when you have to insert two figures at the front of a manual?) of each. Many other computerized typesetting systems do not allow complete page layout, but this one does. Figures automatically float to a page with enough room for them, footnotes fall on the page on which they are referenced, and if material is added, the rest of the text automatically reformats itself.

Those familiar with typesetting know that any time anything is changed in a book, the effect ripples through the rest of the book, often requiring intervention by the typesetter on each page. Because things move around, technical writers have learned never to refer to other sections by page number; with TEX, this is no longer the case.

The heart of the HPTEX system is the HP2688A laser printer, which is no bigger than a computer terminal. The resolution produced by this little marvel is 300 dots per inch. While that does not approach the 3000 dots per inch of the Autologic APS-5, it is considerably better than most dot matrix or daisy-wheel printers. And quieter. And faster.

And For The 1000

Third-party vendors provide TEX for other HP computers, including the 3000 and the 1000. On the 1000 minis and micros with RTE-6/VM or RTE-A operating systems, JDJ Wordware offers **TEX/1000**. You need at least 512K main memory and 2Mb disk storage. Output devices include the Toshiba P1350 and Epson MX80, with HP2688A soon to be supported. TEX/1000 formats pages at 5 to 30 seconds each, depending on processor and memory. The output driver builds a page in memory, and then dumps the graphics to a printer. $2500.

Tick Tock

TYX Corporation offers an implementation of TEX, written in C rather than the usual Pascal (or SAIL), on DEC Rainbow, Victor 9000, IBM PC, and their own integrated **TYXSet 1000** system. The latter consists of a minicomputer that supports up to 24 users, a Canon Laser printer, a Mergenthaler Omnitech laser typesetter, terminals, and a typesetting interface. In fact, this system could provide the entire front end for a typesetting system suitable for a magazine or the publications department of a medium-sized company. No longer would a magazine have to wait several days to look at galleys and several more for the corrections.

TYX also offers smaller versions of their system configured on DEC Rainbow, Victor 9000, or IBM PC. All come with software and a friendlier interface than normally comes with TEX. Various function keys insert commands as needed, including the right number of beginning and ending group delimiters. All systems include a screen preview mode. While this implementation is TEX78, TYX expects to have TEX82 within a year. TYX also provides the service of taking TEX output files from customers who do not have phototypesetters and providing camera-ready copy from their own Autologic.

Get on the TUG

If you plan to use TEX, or attempt to implement it on your system, you will want to talk to others who have already done it. TEX has been put onto everything from Z80 systems to IBM 370s. The **TEX Users' Group** puts people in touch with "site coordinators," those in your area who have already implemented TEX with a system like yours. They also publish *TUGboat*, have yearly meetings, sell TEX tee shirts, etc. They'll send you an information packet containing *TUGboat* 1 #1, "TEX and Metafont: Errata and Changes," the TEX Users' Group Membership List (with the over 800 members sorted by name, computer, and output device), and forms for ordering TEX82 and joining TUG. You can also get information on videotaped instruction and special in-house courses on TEX.

Lost of Storage

Optical disk storage may be available within a few years for most micros, and at the prices now being paid for hard disks. The advantage is storage not in megabytes, but in *gigabytes*! Companies like Optimem, Thomson CSF, Panasonic, and 3M are

working in this field. Who needs that kind of storage, you ask? Hospitals, storing thousands of digitized x-rays for immediate display, each of which can require 512 by 512 pixels by 16 bits deep, CAD systems, and movie production companies, who need to store millions of frames with virtually instantaneous recall. Did you know that credit cards are *now* available that can hold megabytes of information? And here's an ominous development: the military is very interested in optical drives because they feel such drives to be the only medium that could survive a nuclear attack. You can read all about all of this in *Optical Memory News*, $295 per year for six issues.

Improving the Compatibles

The **STM Personal Computer**, from STM Electronics (the folks who brought you the Pied Piper portable), a 17-pound, IBM PC-compatible, transportable computer based on the 80186 true 16-bit CPU, with integrated 16-line, 84-character, LCD, backlit by an electroluminescent panel for total darkness use, 40-column thermal printer, 94-key (including numeric pad and 10 function keys) keyboard, programmable 300- to 1200-baud auto dial/auto answer direct connect data communications modem, 256K RAM (expandable to 512), dual 1Mb (800K each formatted) floppies, standard video monitor support with windowing software and high resolution graphics, parallel port, two programmable serial ports, hard disk interface, edge connector to send I/O to an external expansion box, and integrated software, running under MS-DOS 2.0, with word processing, spreadsheet, database, graphics functions and communications software all standard, costs $3000, with a desktop version for $2500.

Cross Assembling 68K

Quelo wants us to know that their cross assembler for the MC68000 runs not only on CP/M-80, but also on CP/M-86, CP/M-68K, and PC-DOS, and they offer portable C source code. You get up to 31 characters in symbols, lowercase distinction, byte object linking, XOR, MOD, and NOT, *break* and *elseif* for control structures, error messages in English, a linker with limitless program size and number of modules that can be linked, and linker resolution of complex expressions.

Light Up Your Screen

Tech-Sketch Light Pens, for Commodore, Apple, and Atari, control the cursor or select menu options without hardware modifications. The **LP10-S** requires screen contact, while the **LP15-S** is a high-resolution pen that operates up to six inches from the screen. From $39.95.

■■J

(Continued from page 96)

Microsoft's MS-DOS linkage format supports long symbols (31 unique characters) and has case sensitivity. Thus, it overcomes the objections that Mr. Clark posed for the 8080 .REL system. However, one nagging problem still remains: the dichotomy between object modules and libraries. Because of this dichotomy, the MS-DOS linker always includes the full contents of an .OBJ module during linking.

However, libraries are searched. To make a library efficient, each function must be compiled into its own .OBJ file. Each .OBJ file then becomes a single sub-block in the library, and all the functions of that sub-block file are included at program linkage. If libraries and object modules were equivalent, this problem would be overcome since the functions of an object module would be separable during linking.

Runtime Libraries

Nonstandard runtime libraries are a plague to programmers. They inhibit portability and introduce bugs when software is transported between different compilers. Charles Brady of New South Wales, Australia, writes:

"An enormous contribution to standardization of C programs would be the publication of a standard I/O library for BDS C, with, if necessary, a modified runtime package. The very fast and efficient compiler, particularly with the symbolic debugger tool, provides a very inviting environment for software development. It is a great pity that this means abundant nonstandard C. As there is no inherent reason why this should be the case . . . someone should be able to . . . produce a Unix compatible I/O library."

This point is especially well taken in view of the large amount of BDS C software available through the C User's Group of Yates Center, Kansas.

In the case of the runtime library itself, a clear standard exists. This standard is spelled out in *The C Programming Language*. All compilers have to do is support a proper subset appropriate to the environment in which they work.

Conclusion

In this column, I have included the follow-up discussion on Unix and on link formats/runtime libraries. The discussion has been mainly of a critical nature. However, this is not intended to be a condemnation of the systems but rather an impetus for readers to suggest new ideas to improve what we already have. Unix is a worthwhile standard and should be supported. C is a valuable tool but can be improved. This column is a forum where we can discuss, develop, and nurture new ideas about C and Unix. ■■J

(Continued from page 10)

a Z80 program, is remapped. Where in a real Apple the various I/O cards are memory mapped beginning at C000h, the Z80 sees them at 6000h in bank 0. So what you have to do is to find out which memory-mapped addresses your Apple serial card is supposed to appear at (somewhere in C0*xx*h). Under CP/M Plus, it will appear at 60*xx*h in bank 0.

Now all you have to do is find a way to read and write those bytes of bank 0 from a program running in bank 1. Mark Howard has written a Resident System Extension (RSX) that will do that for you. If it has been installed, you will be able to call it with a bank-0 address and get back the value of, say, the status port of your serial card. You write a MODEM7 overlay to make these calls and, voila, you are on the air. Telescoped into that "voila," of course, are about thirteen hours of sweat and hair-pulling.

Our April Item

We are in receipt of a letter from one T. Heintern of Milpitas, CA. Despite his slightly stilted English (Heintern is an immigrant from Islandia), we think his story will intrigue you.

"Your readers might be interested in the activities of two of my co-workers, or really ex-co-workers. I am, and they were, employed by a well-known maker of magnetic media here in Silicon Valley.

"At any rate, my ex-colleagues recently left our beneficent company and struck out on their own. They were remarkably close-mouthed about what they meant to do, but everyone in our department was aware that they had been spending long lunch hours in the bar at the Red Lion Inn in San Jose, in the company of various folk who wore the kind of urban-western gear that replaces a venture capitalist's ivy-league suit when he aims to blend into the crowd 'out west.' I presume my peers found the money they needed, or at any rate the strong hint of it, because they tendered their resignations just before Comdex.

"I next saw them in Las Vegas, where they were holding some fairly lush parties in a private suite. It was also at Comdex that I began to pick up rumors of a new, even-smaller, floppy disk format that (everyone told me) was about to burst on the scene. 'Forget the three-inchers,' I heard one fellow tell another in the midst of the crowd at the Microsoft Windows demo. 'It's almost standardized, so it's dead, right? The big thing is gonna be the CPS . . . .' Here the crowd pressed me away from them. Had they said CPS? I wasn't sure. Later I heard similar references to a 'Colt.'

"Back home I was swapping rumors with the other folk who'd attended the convention, and they too had heard rumors of a new diskette – and that it was to be of domestic, not Japanese, manufacture. Best of all, one had picked up an indication of its size: it was to be of 3.8 cm diameter.

"Well, I was mulling all this over when it all fell into place in my mind: CPS, Colt, 3.8 cm – a Colt Police Special is a 38-calibre pistol! Just as 'Winchester' was the code name for the IBM 3030, 'Colt' was a code name for a 3.8 cm diskette. Three point eight centimeters comes to about one and a half inches, I thought. And the light dawned!

"There flashed before my mind's eye the picture of our production floor: The great sheets of mylar spinning off the drums and under the punches . . . the thousands of 3.8 cm mylar circles fluttering down in drifts from the 8-inch floppies-to-be . . . our janitors sweeping them up . . . our teams of diligent housewives bundling them into tidy stacks to be piled in building-high columns in the insulation space of our great Eastern wall.

"I dashed outside and rapped on the wall – hollow! Our insulation, years of 3.8 cm floppy-punchings, was gone!

"I have, of course, notified corporate management, who are taking legal advice. However, I wish to warn your readers. Shun the new diskette format! After all, you've already paid for the doughnuts. Will you now buy the holes?" ■■J

Sixth Generation Computers

Dr. Dobb's Journal

For the Experienced in Microcomputing

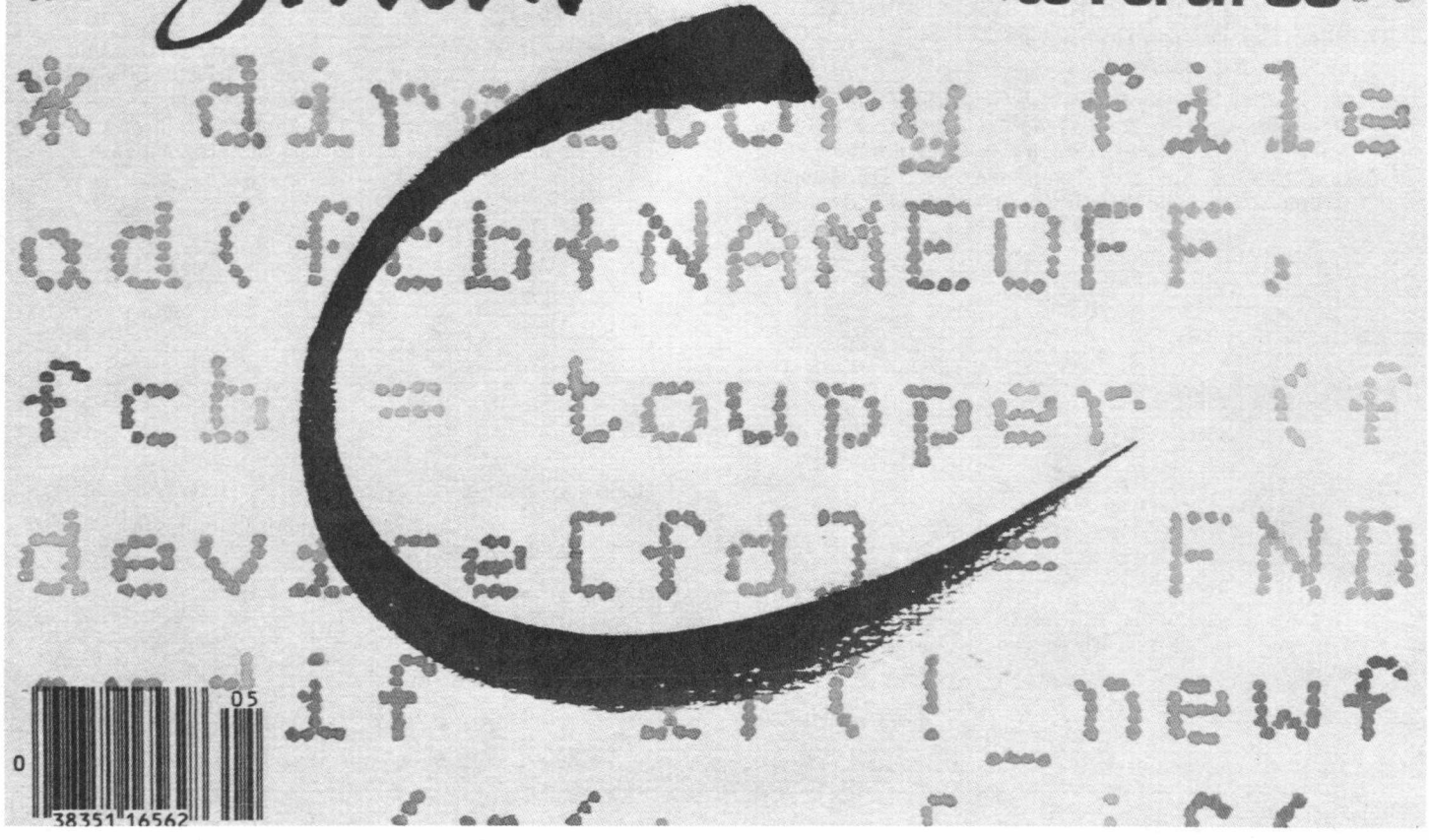
#91 MAY 1984

$2.95 [3.50 Canada]

A New Library for Small C

Introduction to Modula 2

Converting Fig-Forth to Forth-83



0 05 38351 16562

Dr. Dobb's Journal

For the Experienced in Microcomputing

May 1984
Volume 9, Issue 5

CONTENTS

ARTICLES

22 Introduction to Modula-2 for Pascal Programmers
by Hugh McLarty and David Smith

Modula-2, Niklaus Wirth's latest language, has been compared to Pascal and even Ada. Many programmers may have been eyeing this successor to Pascal. This article compares features of the two languages, giving Pascal programmers an idea of what to expect in the Modula environment.

28 Converting Fig-Forth to Forth-83
by Ray Duncan

In mid-1983 the Forth Standards Team adopted a new Forth standard, Forth-83. While this new standard provides a number of improvements, its incompatibilities with older versions of Forth can cause problems when trying to run old programs. The author (on brief sabbatical from his 16-bit Software Toolbox) discusses how to convert old Forth programs to Forth-83, including a checklist of things to watch for and a summary of the 83-Standard Forth vocabulary changes.

37 Sixth Generation Computers
by Richard Grigonis

The author has addressed Fifth Generation topics in previous issues of *Dr. Dobb's Journal* (December 1982 and August 1983). Now he turns to even more distant speculations by describing how the problem of increasing computational speed could be approached in the Sixth Generation.

50 A New Library For Small-C
by James Hendrix and Ernest Payne

In December 1982 and January 1983 we published James Hendrix's version 2.0 of the Small-C compiler, which contained a relatively meager library. This month we present a more sophisticated library incorporating several new Unix-like features, along with code that upgrades the compiler to version 2.1. Because of the size, the last part of Listing Two will be continued next month. **(Reader**

82 The Accent Finder
by Eddy Vasile

Many of us rely on sound and intuition for things like proper syllabic division of words, but computers must use more defined methods. Written as an exercise in the use of sets in Pascal, this program breaks Spanish words into syllables and determines where accents are to be placed.

98 Solutions to Quirks in dBASE II
by Gene Head

While dBASE II is very powerful, it does have some shortcomings. This is the first of several articles dealing with fixes to some of its quirks. Here the author shows how to add an INKEY function, allowing interruption of a long listing or printout without aborting the program entirely.

DEPARTMENTS

6 Editorial

9 Letters

12 Dr. Dobb's Clinic

Optimizing Compilers; Timeless Language (Prolog); Prissy Program?; I Tell You Three Times; Computer Citizenship

16 CP/M Exchange

IRD: No Longer A Mystery; CP/M Tidbits; OUT: and INP: – Making Them Work

100 Of Interest

Editor-in-Chief: Michael Swaine
Editor: Reynold Wiggins
Managing Editor: Randy Sutherland
Contributing Editors: Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg
Copy Editor: Polly Koch

Director of Operations: Beatrice Blatteis
Promotions Coordinator: Jane Sharninghouse
Circulation: Sally Brenton
Advertising Sales: Alice Hinton, Walter Andrzejewski

Production/Art Director: Shelley Rae Doeden
Production: Alida Hinton
Typesetting: Jean Aring
Cover Illustration: William Stewart

Otmar Weber
Publisher
Chairman of the Board
M&T Publishing, Inc.

C. F. von Quadt
Director
M&T Publishing, Inc.

Laird Foshay
President
M&T Publishing, Inc.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto CA 94303, (415) 424-0600, *under license from People's Computer Company*, Box E, Menlo Park, CA 94626. Second class postage paid at Palo Alto and at additional entry points. Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto CA 94303.

Subscription Rates: $25 per year within the United States $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R. B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282,Flugelstr. 47,4000 Dusseldorf 1, West Germany; Euro Computer Shop, 182 Rue du Faubourg St. Denis, 75010, Paris, France; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014, Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** AXCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia, Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian Distributors)*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

EDITORIAL

Changes come and trip all your traps. You retire to write that novel and you find yourself behind a desk again, doing calisthenics with metaphorical braces on your teeth. Life's like that.

It's like this.

This magazine began over eight years ago as *Dr. Dobb's Journal of Tiny BASIC Calisthenics and Orthodontia*, its slogan being "running light without overbyte," and its notion being to promote the kind of programming gymnastics that didn't waste a byte of then-precious memory. It was a good notion, and the best programmers on microcomputers gladly shared their insights on the pages of *Dr. Dobb's* without monetary remuneration. Over the years the magazine continued to distribute, discuss and promote exemplary software: Tiny BASIC made way for Tiny C, a miniature version of Ada and other virtuoso performances in computer calisthenics and orthodontia.

But changes come. Within the past year, *Dr. Dobb's* has started paying its contributors, expanded its editorial staff and become affiliated with M&T Publishing, a profit-making company. It seems fair to ask how much of the original spirit of the publication can survive such changes.

It's a question whose answer I care about. I've been reading and enjoying *Dr. Dobb's* for years, and recently, in the process of researching a book on the history of the microcomputer, I read all the early issues of the magazine. I came to love *Dr. Dobb's Journal* for its irreverence, its personality, its exuberant embracing of tough problems for their very toughness. I don't want it to lose those qualities.

It's not hard to imagine some sad scenarios for *Dr. Dobb's*. It abandons 8-bit software as obsolete and becomes a machine-specific magazine dedicated to some well-known 8088-based personal computer. It ignores any advances in computer architecture since 1979 and lives out its days recycling IMSAI technical notes. It "targets the home [or small business or knowledge worker] market" and disappears in the newstand collage. It fancies itself *Time* magazine and puts red borders on its covers.

Happily, I can promise that none of those scenarios will come to pass in the forseeable future. Who I am is Mike Swaine, former Senior Editor at *InfoWorld*, and as of this issue, Editor-in-Chief of *Dr. Dobb's Journal*. My job is a new one; Renny Wiggins is still Editor. Renny promises, too.

Changes come, and significant changes are ahead for *Dr. Dobb's*, but changes, I think, that respect the magazine's traditions. You can expect the Doctor to experiment with some techniques (like an electronic bulletin board) for getting his methods of software distribution into the 1980's, but don't imagine that he'll abandon his interest in cheap, public-domain software. Expect coverage of the Macintosh computer, not because it's popular, but because good programmers are starting to do amazing things with it. Expect to see some industry leaders in the pages of the magazine, but not at the expense of new writers with something to say.

Expect to see coverage of good implementations of languages and system software and utilities, and occasionally some really unusual things. For example, this month, Renny has put together a rather orthodontic issue, including an introduction to Modulo-2 for Pascal programmers, Ray Duncan's piece on converting to the Forth-83 standard and an unusual article by Richard Grigonis of Children's Television Workshop on what he calls "sixth generation computers."

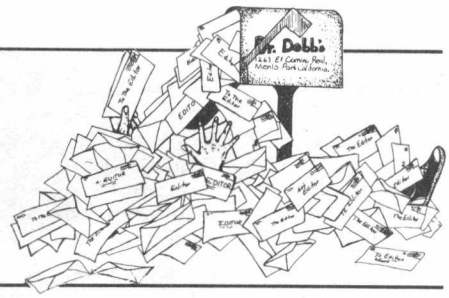
Next month we'll examine a much-discussed and much-purchased piece of software, Borland International's Turbo Pascal. If half the things that are claimed about it are true . . . but we'll find out next month.

Mike Swaine

This Month's Referees

David Cortesi, Contributing Editor
David Clark, Pennsylvania State University
Clay Phipps, ACM
Henry Socha, SochaLogical Research

LETTERS



The editorial response card is a great way to talk to us, but don't forget that Dr. Dobb's Journal *also welcomes letters to the editor as a forum for ideas, innovations, irascibility and even idiosyncrasies. Some letters may be edited for clarity and brevity. The Doctor likes hearing from you – keep on writing.*

Ackermann Elaborated

Dear Editor:

I am writing in response to the letter from Mr. Paul Condon (*DDJ* No. 88, February 1984). To logicians, Wilhelm Ackermann is best known in the context of Hilbert and Ackermann, *Mathematical Logic*, 1928. In 1926 David Hilbert asked whether all computable functions are primitive recursive. Ackermann found the answer in 1928, with Ackermann's function. (A translation of Ackermann's original article may be found in van Heijenoort, *From Frege to Gödel*, Harvard University Press, 1967. Ackermann, W., "On Hilbert's Construction of the Real Numbers.")

Primitive recursive functions are, roughly, those that can be computed from the simple functions of sum, product, power, and the operations of substitution, composition, and induction upon these. Ackermann's function is clearly computable, but can be shown to "increase" more rapidly than any primitive recursive function. (A good reference is Hans Hermes, *Enumerabililty, Decidability, Computability*, Springer-Verlag, 1969. Also many texts on formal language theory discuss Ackermann's function.)

The interest in the question of whether or not there exists a recursive, but not primitive recursive, function lay in determining if the mathematical definition of primitive recursion corresponds to our intuitive notion of "function computable by algorithm." Ackermann's function shows that primitive recursion does not; Church's Thesis, almost universally accepted, is the view that the broader class of recursive functions does correspond. There can be no exact mathematical demonstration that some class of mathematically definable functions does correspond to our intuitive ideas of computation, since the latter are necessarily imprecise.

Recursive functions expand the class of primitive recursive functions by allowing for the calculation of the least y such that $f(x=y)$ in, for example, the domain of natural numbers. Ackermann's function can fairly easily be seen to be recursive; it can be calculated from the primitive recursive functions with the addition of this minimization feature. What this shows is that the induction in the definition of Ackermann's function is not "ordinary" induction.

Perhaps the minimization feature of recursive functions is what Mr. Condon is getting at with his concluding remark about the correctness or incorrectness of certain definitions of Ackermann's function. I haven't seen the definitions in question.

Beyond the testing of recursion in computing languages that he mentions, the only "use" that I know for the function is the theoretical one of categorizing classes of recursive functions. It is quite likely that Ackermann's function has some wider applications in recursion theory of which I am unaware; I can envisage applications in the study of proof theory for infinitary languages, via Gödel numbering, or just in infinitary proof theory. It sure is fast, isn't it?

Sincerely,
Jay Halcomb
Philosophy Department
University of Arizona
Tucson, AZ 85719

Dear Editor,

In a recent letter, Paul Condon inquired about the origin of Ackermann's function and asked of what use it is. Well, the story goes back to the mid 1920's when Wilhelm Ackermann was working with David Hilbert on an ambitious program to put the foundations of mathematics on a firm axiomatic footing. Among several topics they studied was a class of arithmetic functions, defined on the nonnegative integers, which they called "primitive recursive functions."

An exact characterization of the primitive recursive functions is a little too technical for this note. Roughly speaking, however, they are anything you can get by starting with just constants and the successor function $S(n) = n + 1$, followed by rules of composition that allow one to plug one function into another as an argument and to define new functions by induction through the scheme $f(n + 1) = g(n, h(n))$ where g and h are functions previously defined by the same rules.

It turns out that the primitive recursive functions form a very large class. At that time it looked as though any function at all that was defined on the natural numbers and for all values of its arguments might in fact be primitive recursive. This question was raised by Hilbert in 1926. In particular, he asked: Can recursion be used to define a function that is not primitive recursive? The answer was given in 1928 when Ackermann came up with an example of such a function. But the original Ackermann function was not the same as the version that is currently fashionable. The original function had three arguments instead of two. The form currently popular (as quoted by Condon) is apparently due to Rózsa Péter in 1935, although she may have published earlier in some obscure Hungarian journal.

$$\begin{aligned} Ack(i, j) = {} & \text{if } i = 0 \text{ then } j + 1 \text{ else} \\ & \text{if } j = 0 \text{ then } Ack(i - 1, 1) \text{ else} \\ & Ack(i - 1, Ack(i, j - 1)) \end{aligned}$$

Using Péter's form, suppose one defines a new function A of one argument by putting $A(n) = Ack(n,n)$. Then it can be shown (but not in this note) that the function A(n) grows more rapidly with n than any possible primitive recursive function of one argument, and therefore that A is not itself a primitive recursive function. This is what Ackermann proved and what his function was designed to illustrate.

The growth rate of A(n) is really mind-boggling. For comparison, consider the function B(n) = n!!!...!! where there are n occurrences of the factorial sign. Now B(n) is primitive recursive and its first few values are $B(0) = 0$, $B(1) = 1! = 1$, $B(2) = 2!! = 2$, and $B(3) = 3!!! = \ldots$ well, B(3) has 1747 decimal digits so I won't print it here. So B(n) gets out of the starting gate fast and grows at such a fantastic rate that if the entire universe were filled with high-density floppies they wouldn't be sufficient to store the value of B(4). On the other hand, the value of A(3) is only 61, so A(n) gets off to a slower start. But A(4) is already enormously greater than B(4), and A(5) makes B(5) look like an infinitesimal quantity.

The fact that Ackermann's function is defined by a double recursion makes it useful as one kind of check on correctness of the implementations of recursion in language interpreters and compilers. However, a recursive form of definition, of course, does not mean that a recursive language is necessary to evaluate a function (although it is sometimes not obvious how to do so in a nonrecursive manner). I

offer two little Pascal functions, ack1 (Listing One, below) and ack2 (Listing Two, below), each of which computes the same values as the recursively defined Ack(i,j) discussed above. Neither program uses recursive calls, so they easily can be translated for execution into languages such as BASIC. The function ack1 operates by "faking" an evaluation stack. The function ack2 is more mysterious because it uses so little storage and runs so much faster than the function ack1. The interested reader might want to figure out what is going on in these programs by inserting print statements that show the contents of the arrays after each iteration.

Sincerely,
Milton W. Green
440 Sherwood Way
Menlo Park, CA 94025

Telegovernment?

Dear Editor,

We are in a period where many voters feel their vote means nothing. They do not feel their elected representatives care what their constituents want. As a consequence more and more people have stopped voting.

Over the years the political control of our nation has become ever more centralized in the federal government. At the same time the population has become more decentralized politically. Who is your representative? Does your representative make any effort to keep up to date with the consensus of opinion in his/her district? What methods does your representative use to see that the people he/she represents know what he/she is doing?

Today, we have the basis for alternative methods of participation in government, as well as in other political activities. Computer-based telecommunications combined with television, either public or commercial, can provide the communications links required to return government to the people. An ever increasing number of families now have a computer or terminal in the home.

The time to start taking action regarding the application of telecommunications to government is now. By the time requirements and procedures have been decided we will already have enough users to implement any determinations made. It's also important to prevent, through early action, perversion of the potential power of political telecommunications by power groups.

One method of implementation might be to require every representative to maintain a telecommunications computer as part of his office equipment. This would be accessed from his district through an 800 number, the maintenance of which could be a part of the requirements of doing business, as a local or national monopoly, for the telephone companies.

We have entered a period of technology where government in the United States can once again be of the people, for the people, and by the people. The development of television as an educational medium, teletext, and computer-based telecommunications makes this potentially possible. The continuing reduction in hardware costs makes it economically viable.

This proposal does not advocate an end to representative government at any level. Instead it would return the same control to the electorate enjoyed by the electorate in 1776. We would still need our present legislative, judicial, and executive institutions as they are now constituted; direct democracy allows no check or time delay on temporary and emotional public reaction.

It's necessary for us to implement special education on various policy problems. The need for national "town meetings" has reached a critical level.

It is my wish to start a national organization to investigate, analyze, discuss, and possibly start implementation of public participation in government through telecommunications, television, etc. Anyone interested in participating should feel free to contact me. It will be difficult for me as an individual to provide the initial start of such an organization. This is based on both time and finances available, but I will endeavor to do my best.

Yours truly,
D. R. Crago
P. O. Box 728
Placentia, CA 92670

■■J

Letters (Text begins on page 9)

Listing One

```
function ack1(i, j: integer): integer;
    var p: integer; a: array[0..4000] of integer;
    begin
        a[0] := i; a[1] := j; p := 1;
        repeat
            if a[p-1] = 0 then begin
               a[p-1] := a[p]+1; p := p-1 end else
               if a[p] = 0 then begin
                  a[p] := 1; a[p-1] := a[p-1]-1 end else begin
                     a[p+1] := a[p]-1; a[p] := a[p-1];
                     a[p-1] := a[p-1]-1; p := p+1 end
        until p = 0;
        ack1 := a[0]
    end;
```

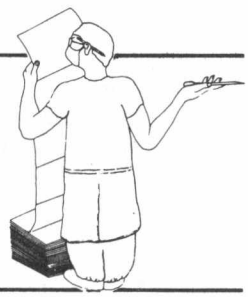
(End Listing One)

Listing Two

```
function  ack2 (i,j: integer): integer;
    label 1, 2, 3;
    var p: integer; a,b: array[0..6] of integer;
    begin
            for p := 0 to 6 do begin a[p] := 1; b[p] := -1 end;
        1: b[0] := b[0]+1; a[0] := b[0]+1; p := 0;
        2: if (i = p) and (j = b[p]) then goto 3;
           if b[p] <> a[p+1] then goto 1;
           p := p+1; b[p] := b[p]+1; a[p] := a[0]; goto 2;
        3: ack2 := a[0]
    end;
```

(End Listings)

DR. DOBB'S CLINIC



by D.E. Cortesi, Resident Intern

Optimizing Compilers

We asked for help finding a reference to what we thought was the first optimizing compiler, and we got some good responses, of which the most notable came from C. W. Medlock:

"I believe you are referring to IBM's Fortran H compiler for the IBM 360/370. As to the reference to this work that you are seeking, I believe that you are referring to our paper, "Object Code Optimization," *Communications of the ACM*, Vol. 12, No. 1 (January 1969), written by myself and E. S. Lowry.

"It would be unfair, however, to claim that we were co-authors of the *first* optimizing compiler. Indeed, credit belongs to such early pioneers as John Backus and his original crew for the fine job they did in generating the most-efficient code that the early versions of even IBM 704 Fortran turned out. Many think that it was the quality of this code (compared to that produced by earlier compilers) that convinced the computing community of the feasibility of using high-level languages at all!

"While we know of no other full use of these techniques in another compiler (and quite frankly wish we could find such a compiler for the C language for the IBM PC), there is at least one company, Computer Innovations, that claims to have such a compiler. We haven't been able to test it to see just how 'optimized' the code it produces really is.

"As to the optimization technique that Ed Lowry and I developed, it relies heavily upon tracing the flow of both data and control within a program to see which code items can be moved to a less-frequently-traversed portion of the program (or eliminated entirely as either redundant or producing output that would never be used), and to assist with the temporary assignment of variables to registers over the most frequently executed portions of the code. This analysis was performed over the entire compilation and the technique was thus called 'Global Program Optimization'."

Another reader, Owen Montgomery, also wrote to point out the importance of the early Fortran compilers. He says, "Fortran I was the first algebraic optimizing compiler. At the inception of the Fortran project by John Backus's team at IBM in 1954, most programming was done in machine code or simple assemblers. So-called automatic coding systems existed but would be regarded as little more than symbolic assemblers with built-in function libraries by today's standards. Fortran, to be successful, had to produce very efficient object code. If not optimum and on a par with hand-coded programs, it would surely fail. The consensus among programmers of the day was that efficient programming could not be automated."

Owen's letter included an illuminating quote from a paper by Backus, "The History of FORTRAN I, II, and III," *ACM SIGPLAN Notices*, Vol. 13, No. 8, August 1978. Read it carefully:

"It was an exciting period; when later on we began to get fragments of compiled programs out of the system, we were often astonished at the surprising transformations in the indexing operations and in the arrangement of the computation which the compiler made, changes which made the object program more efficient but which we would not have thought to make as programmers ourselves . . . . Transfers of control appeared which corresponded to no source statements, expressions were radically rearranged, and the same DO statement might produce no instructions in the object program in one context, and in another it would produce many instructions in different places in the program."

A very few times we've had the experience of being delighted by the clever object code a compiler generated, but never on a personal computer. On the contrary, the personal compilers we've used have produced the dullest, most predictable, most flat-footedly literal translation one could imagine.

The entertainment value of the code is not the issue, of course. The crux of the matter is that a genuine optimizer can produce the kind of low-level, foxy cleverness that an assembly programmer sweats to invent – and does it by mechanical methods. That liberates the programmer; the brain power he or she would have spent on achieving tight code can instead be applied where it really matters: on the algorithm.

What we'd like to know is, if Backus could accomplish this in 1954; if Lowry and Medlock could do it in 1969; if the techniques can be found in any of a dozen computer-science textbooks; *why don't we have such compilers today?*

Timeless Language

Meanwhile, we've been exploring the other end of the language spectrum. As part of a continuing (and probably futile) attempt to keep up with the state of the art, your Intern has been studying PROLOG. That's the language that figures heavily in the Japanese plan to produce the Fifth Generation of computers. It is, effectively, an executable variant of the notation of format predicate logic.

At the moment, most implementations of PROLOG are handcrafted prototypes running in university computer centers, primarily in Europe. However, one implementation is a real, commercial product. Known as micro-PROLOG, it's published by Logic Programming Associates Ltd. (10 Burntwood Close, London, England SW18 3JU; $275, shipping included). We recommend it. The manuals are more than adequate; the software works; and the package was delivered across half the world faster than some we've ordered from across the state.

Micro-PROLOG is a fascinating piece of software in itself, entirely aside from the intrinsic fascination of an entirely new language. At its core is a small, fast Z80 interpreter for a list-processing language, one that is strongly reminiscent of LISP, but augmented with pattern-matching primitives. This implements the core of PROLOG, because the core of a PROLOG interpreter is a recursive, backtracking pattern-matcher.

The rest of the system is implemented in PROLOG. This is impressive, because "the rest" includes everything that makes the system usable, including an interactive front end that hides the LISP-like parenthesized notation under a more natural syntax. This gives one confidence in two ways: it demonstrates that PROLOG is a "real" language, since it can be used to write non-trivial software; and it affords proof that the core system is well-tested, since it can execute its own front end.

PROLOG is something else again, it requires the same kind of head-spinning reorientation that accompanies a move from, say, Pascal to Forth, or from COBOL to APL. The most difficult part, at least for *this student*, *is that* PROLOG contains no concept of *time*. Every other language we've used contains an implicit notion of movement in time: the computer will do this *and then* it will do that.

In PROLOG, every statement of a program is a candidate for execution at every moment. Actually, "statement" is a poor term for a programmer to use in describing a line of PROLOG; it carries too many implications from other lan-

guages. "Assertion" is a better term; a PROLOG program is composed of a number of assertions about the world. In principle, all assertions are executed at once, in parallel, and those that turn out to be false are discarded. (In practice, the interpreter tries them sequentially, but it isn't cricket to assume that.) It's this inherent parallelism that makes PROLOG a candidate for the system language of the Fifth Generation machines, but boy! does it twist your mind at first.

The micro-PROLOG system includes a hands-on tutorial that serves to get you just barely started. Two other books may help. *Programming in PROLOG* by W. F. Clocksin and C. S. Mellish (Springer-Verlag, 1981) is a reasonably good textbook on the use of the language. *Logic for Problem Solving* by Robert Kowalski (North-Holland/Elsevier, 1979) is an exhaustive, scholarly exploration of the relationship between Horn-clause logic (which PROLOG implements) and everything else in the world. It is heavy going, and parts are interesting only to an implementor of PROLOG, not to a user of it. Still, parts of Kowalski's book explain the rationale behind this initially puzzling language.

One warning: the syntax of PROLOG is nowhere near standardized. Micro-PROLOG seems to be full PROLOG as described by Clocksin and Mellish (it may even be a superset). However, the syntax of micro-PROLOG has many differences of detail from that used by Clocksin and Mellish, and that in turn is different from the syntax that Kowalski uses. It is not difficult to translate from one to another, but it imposes an extra mental load.

Are any readers also exploring PROLOG? Somebody must be, because our local bookstore can't keep *Programming in PROLOG* in stock. Write and share your experiences.

Prissy Program?

Oscar Goldman writes to ask, "Did you know that RMAC is a delicate flower who turns her (his? its?) head in embarrassment at the presence of an error? Take a look at this listing:

```
0000  3A0600   lda  store
      inra     ; typographical error
0003  320600   sta  store
0006  00  store: db  0
      end
```

It took me quite a while to debug a program which had just that typo (of 'inra' for 'inr a'). I think you might warn your readers of this one."

With pleasure, Oscar. What we have here is the result of the peculiarly flexible syntax of the Digital Research assemblers.

Typical assemblers require a label to begin in column 1 of a statement. That permits a simple test for the presence of a label: if column 1 is blank, there isn't one. The DRI assemblers (ASM, MAC, RMAC) have a more complicated rule. They isolate the first word of a statement and look for it in the table of known operation codes and in the table of defined macro names. If the first word is not an opcode or a macro, it must be a label.

That's what happened here: since "inra" isn't an opcode, RMAC decided it must be a label. It shows up in the symbol table listing. The fact that it was indented to line up with the other opcodes made no difference at all.

This syntactic scheme can be useful; you can define pseudo data structures, indenting inner names under the names that contain them. However, as Oscar notes, it can also let you create some very tricky bugs.

I Tell You Three Times

Another correspondent, Michael Barr, writes with a tale of keyboard handling in MSDOS 2.0. We don't have an MSDOS machine on which we can test his claims, but perhaps a reader can test and explain them. Here is his account.

"I have discovered a peculiar effect of control-S. To see it, at the system prompt simply type control-S. Do it again, then a third time. Try it three more times. Now press control-S, followed by any ordinary character, then control-S again. Confused? So was I.

"There are two DOS functions that are relevant. The first is function 7, which goes into a loop until you type a character and then returns that character. It works as documented. The second is function 11, which is documented to return a true flag only if there is a character waiting in the keyboard buffer. What it actually does is another matter.

"After the first control-S, that character is in the buffer (as can be determined with either function 6 or function 7), but function 11 swears the buffer is empty. After another keypress, whether control-S or another character, assuming you have not gotten the first control-S out of the buffer, the buffer is *empty* while function 11 returns a *true* flag.

"What happens after the third keystroke? Well, that is exactly like the first. Why, then, do you actually get a character that third time? I can only speculate that the command interpreter in DOS operates in the same way as a program of mine. What my program did was to call function 11 until returned a true flag, then call function 7 to get the character. That way the program could do something else instead of just idling until there was input. (It wasn't doing anything terribly urgent – it was polling location 0000:0417 hex to find out if the Caps Lock key had been pressed.)

"The first entry of control-S had no effect on function 11, while the second caused function 11 to return a true flag with the buffer empty. Then the program called function 7, which just idled until another character was entered. Thus every third keypress was captured. I speculate that the DOS command interpreter is doing likewise."

We can't check out Barr's problem, but here's a hint for any reader who wants to look into it. It's suggestive that the problem centers on control-S. That's XOFF, or DC3, and in many systems it would be a signal to freeze screen output until a control-Q (XON, DC1) was entered. MSDOS contains a number of half-assimilated CP/M compatibilities (Barr also mentions the undocumented feature that control-P will toggle printer echo of the screen output, which is a CP/M convention). CP/M 2 uses control-S to freeze the screen, releasing the screen when any further key is hit. Both keystrokes are swallowed in the system. Maybe that's what MSDOS is trying to do, but not thoroughly enough.

Computer Citizenship

However, another point in Barr's letter raises our hackles. In our oh-so-humble opinion, polling loops are bad! A program that refuses to read the keyboard until its poll shows that a character is waiting is a bad citizen; it defeats any number of nice features that the DOS could otherwise implement. It defeats the largest benefits of a buffered keyboard (see this column for May 1983, under the heading "Software vs. Common Sense").

Furthermore, a polling program is absolutely deadly in a multitasking system. It always uses its full slice of time even when it has nothing to do; that defeats the basic idea of multitasking, which is that one task can make good use of another task's idle time. And multitasking systems are coming to micros: Concurrent CP/M is available now, and MSDOS 3.0 is rumored to be one.

Sure, a Caps Lock flag on the status line is a cute feature, but consider: Under Concurrent CP/M, a program like Barr's would be checking the Caps Lock byte at times when the user was entering data to a completely different program. The same problem would probably occur under Microsoft Windows(tm) or VisiOn(tm) – it would be polling Caps Lock when the cursor was in another program's window.

This is an example of how a solution can go wrong when it is implemented at the wrong level of the system. The PC keyboard lacks two 29-cent LEDs to display the state of Caps Lock and Num Lock. The PC operating system does nothing about it. So application programmers try to make up for it, either with a polling loop or by stealing the keyboard interrupt vector. And all the ones whose efforts succeed end up with programs that flat *won't work* in systems where the hardware is a resource that has to be shared among multiple programs. DDJ

CP/M EXCHANGE

by Robert Blum

A few months ago, Terje Bolstad of Norway wrote to ask if I had any experience with an undocumented device he had found in PIP called IRD:. Unfortunately, I had no prior knowledge of the IRD: device to share but was intrigued, as many of you have been, with its origin and possible purpose. I decided to go directly to the mountain for further advisement.

My first letter to DRI was promptly answered. Rather than being provided with an answer, however, I was referred for an explanation to the OEM from whom I had purchased my CP/M system. Since several of us had already done considerable research on how IRD: worked, I didn't feel that a side trip to another vendor was necessary. I wrote to DRI again to explain more fully what steps had been taken to decipher the mystery device's purpose. The response to my second letter came via a phone call, but to no avail because the answer was considered to be proprietary information of DRI.

One further encounter with DRI proved to be no more fruitful than the first two. Undaunted, I took my quest for knowledge to the readers of this column in January. To stimulate the juices of adventure I offered a *DDJ* tee shirt to the author of the best answer.

I should have known what would happen next. However, I was caught off guard when my mail box began to fill with more than my usual quota of correspondence. Your responses to my request for help ranged from a few lines jotted on a postcard to catalog-sized envelopes overstuffed with program listings. As is always the case, and very pleasantly so, when dealing with *DDJ* readers the answers were most insightful and gave evidence to a great deal of work.

It would be impossible to print all the responses that I received. With the exception of some editing, what follows is a cross section of your answers to the IRD: mystery.

IRD: No Longer A Mystery

Dear Bob,

I've just seen your column in the January issue of *Dr. Dobb's*. There really is no mystery about the device called IRD:. The letters "IRD" stand for "Infra-Red Detector." The IRD: device represents a peripheral device whose purpose, clearly, is to monitor the infra-red level in its vicinity. This can be used for various purposes, for instance, to shut down the computer when the programmer's temperature rises above a certain critical level because of CP/M's idiosyncrasies. But the truth, actually, is this: In the beginnings of CP/M, Kildall and associates used to engage in target practice using guns which emitted infra-red (instead of water) and had several IRD's installed in the targets to record their hits.

Oscar Goldman

Dear Robert,

I was so interested in your PIP mystery device that I had to solve the problem before I went to sleep. It seems that the IRD: driver is for the Intellec-8/Icom Reader. On my system PIP hangs (actually it keeps returning 07fh, which never signifies end of file). A very tight routine within PIP is entered and then sometimes never exited when the IRD: device is used. See Listing One (page 18) for the IRD: subroutine taken from PIP version 1.5.

Craig D. Miller

Dear Robert,

Internal Research and Development? Infra Red Device? Irreversible Route to Delirium?

No, the mystery device is, in fact, the Intel Paper Tape ReaDer. Does anyone remember paper tape? Yes, PIP contains a built-in hardware driver for this device which is (was) used with the Intel Intellec-8 MDS system. At least this was true with the V1.4 version of PIP, and I assume the CP/M 2.2 version follows suit.

As for the details, the following actions are performed when each byte is read from the IRD: device during a PIP transfer:

(1) The value OCH is output to port 01H.
(2) The value 08H is output to port 01H. This strobes the paper tape reader mechanism.
(3) PIP loops until bit 5 of port 01H is set to 1. This is the Ready bit.
(4) Finally, the lower 7 bits of port 03H are returned as the input value.

Note that if a system does not contain I/O ports 1 or 3, the net effect of all this will be to return a neverending string of 7fh's, which is what you observed.

IRD: is, no doubt, useless on 99% of CP/M-based systems. In fact, it could be dangerous on systems which use these I/O ports for other purposes.

Now that you know about the Irresistible Riddle to Decode device, you might want to buy an Intellec-8 system in order to realize the full potential of your CP/M software.

Rick Hollinbeck

Dear Mr. Blum:

I think I can shed a little light on the mystery of the IRD: device in CP/M 2.2's PIP.COM. The code appears to drive a strobed parallel input port device. To the best of my recollection, the earlier Intel MDS/Intellec systems have a paper type reader addressed at these port addresses. This suggests that IRD: stands for Intel ReaDer. The reason it performs differently on various systems now becomes clear – it depends on what sort of device (if any) you have addressed at these port addresses. If you don't have any device on port 03h, you will (hopefully) input an FFh which will be masked to 7Fh, which is what you got in your disk file.

This is most likely a remnant of the days when CP/M was still being developed, and that is why Digital Research doesn't like to discuss it. Remember, the original CP/M was primarily written in PL/M, hosted on an Intel development system. They had to read the CP/M utility object tapes somehow, and this is probably how they did it.

Terence M. Kennedy

Dear Robert,

It should be obvious to everyone that IRD: is DRI spelled backwards. However, since acronyms have to stand for something, and Inc. Research Digital sounds really horrible, I propose we christen our device (DRI seems to want to disown the poor child) "Interminable Repetitious Device," in honor of its propensity for generating an endless stream of s's.

I'm sure that if we devote enough collective effort, we can find practical uses for IRD:. In fact, I have already thought of one: By PIPing IRD: to a disk file, we have a convenient, fast way of generating a large file in order to test some program.

David Kettle

Dear Bob,

I suspect that IRD: is simply DRI backwards. If it has to have a meaning, perhaps it would be "Incorporated Research Digitalizer," or some such.

Roy Lipscomb

Dear Sir:

ARD:, IRD:, PRN: – three popular peripheral devices.

This is explained in the documentation for CP/M V1.3, copyrighted by Digi-

tal Research, dated 1976. When the new guide was printed and stapled into a neat booklet the meanings were deleted and PIP.COM was improved. Apparently, PIP's internal table was not completely changed, thus a remnant still exists:

ARD: Addmaster paper tape reader

IRD: Intel or Icom paper tape reader (these two were considered high-speed vs. the ASR-33)

PRN: Tally hard copy printer

There is also a table of error messages in PIP.COM that seem peculiar because I have not seen them reported before: tape stopped or illegal HEX character encountered. The document also explained this in the paragraph about PIP.COM special functions when processing a punched paper tape. I now suspect they are related to ARD: and IRD: as they were never used when I assigned a cassette tape to PUN:/RDR:.

As you may have guessed, I have been an owner of a CP/M operating IMSAI system since November 1976 and, though not having an ASR-33 as a terminal, I did experience punched paper in those historical days.

Delmar Rawson

There you have it, Terje. IRD: stands for Intel-Icom-Intellec-8 ReaDer and was used on that system for reading paper tape. Why all the secrecy about the IRD: device? All we can do is guess, but I am betting that DRI still has an Intel system or two being used daily by one of Jerry Pournelle's mad friends.

Before I forget, my heartiest congratulations go out to Delmar Rawson. He was not only fast but also correct. His *DDJ* tee shirt is on its way.

I would also like to express my thanks to each of you who helped me in finding the answer to the mystery IRD: device.

CP/M Tidbits

Included with Terence Kennedy's response to the IRD: mystery were a few insights into how CP/M does its job:

"In reference to another mysterious topic of discussion in recent months, the reason some programs appear to 'eat' one character before warm booting is because the BDOS itself is reloaded. Let me digress for a moment.

"In CP/M 2.2, there are three sanctioned ways to perform console I/O: via 'normal' BDOS console I/O calls, via BDOS function number 6, and by calling the BIOS directly. In fact, the last two methods do exactly the same thing. I presume Digital Research was preparing us for systems like MP/M and CP/M 3, where accessing the BIOS is a definite 'no-no.'

"Normal BDOS I/O works as follows: whenever a character is sent out to the console, a check is made for an input character being available; it is tested to see if it is a Control-S. Note that this is the only character that the BDOS is looking for at this time. If it is a Control-S, the BDOS enters a tight loop looking for the next character, which may be, for example, a Control-Q to resume output or a Control-C to warm boot. However, if the input character was not a Control-S, it is stored in a one-byte buffer in the BDOS, and the BDOS no longer performs any status checks during output. When the .COM program terminates with a RET to the CCP (as in STAT), the character is still in the BDOS's buffer, and the next BDOS input call will retrieve it first, before calling the BIOS for more characters. If, however, the .COM program terminates with a warm boot, the BDOS is re-loaded, leaving the buffer empty; hence, one character is gone.

"This also causes an unfortunate interaction between BDOS function 6 I/O and the rest of the BDOS I/O calls. Programs should never mix 'normal' BDOS calls with function 6 or BIOS calls. Otherwise, input characters may 'vanish' mysteriously, only to reappear at a later input prompt.

"Digital Research has a stop-gap patch for the latter problem, but only for MP/M II 2.1. They have apparently taken a completely different approach in CP/M 3. In 3, they appear to examine characters as they do in 2.2; but if it is not an appropriate control character, it is discarded. This is why type-ahead routines which worked with 2.2 will not function at the CCP level in CP/M 3.

"Here are two other tidbits that may be of interest to you.

"The CP/M 2.2 'Reset Disk System' BDOS function 13 returns a result parameter in the A register. A is 0 if no $$$.SUB file is active; otherwise it is nonzero. This is how the CCP knows whether or not to continue from the submit file. However, the function only looks at whether or not the first character of any filename is a dollar sign. This is why you may have noticed some strange extra disk activity and a general sluggishness if you have, for example, a filename beginning with a '$' in your directory. It appears that the CCP attempts to open $$$.SUB, fails, and then attempts to delete $$$.SUB.

"The CP/M 2.2 'Raw Console I/O' BDOS function 6 has an undocumented additional option. An E register argument of OFEh merely checks the status of the console, and does not return or delete any pending input characters. This is consistent with raw character handling in CP/M 3 and MP/M II."

OUT: and INP: — Making Them Work

Craig Miller also found the time to include this brief explanation and the necessary assembler source code to implement the OUT: and INP: devices in PIP.COM:

"When DRI upgraded PIP for CP/M Plus, the undocumented IRD: device was dropped and is now only available in V2.2 of CP/M. I have found that the OUT: and INP: drivers are in PIP 3.0; however, the documentation doesn't describe how to install them. The enclosed listing (Listing Two, below) shows how to do this."

■■J

(Listing begins on next page)

CP/M Exchange (Text begins on page 16)

Listing One

```
        MVI     A,0CH           ;Turn reader on
        OUT     01H
        MVI     A,08H           :Turn reader off
        OUT     01H
WAIT:   IN      01H             ;Wait until reader has a character
        RLC
        RLC
        RLC
        RAR
        JC      INPUT           ;If carry, then done (get char), else loop
        JMP     WAIT            ;Loop back
INPUT:  IN      03H             ;Get character
        ANI     07FH            ;Mask parity bit
        RET
```

(End Listing One)

Listing Two

```
;THIS FILE WILL PATCH THE I/O ROUTINES INTO
;PIP FOR VERSIONS 2.2 AND 3.0 (INP: AND OUT:).

TRUE            EQU     0FFFFH
FALSE           EQU     0

PIP22           EQU     FALSE
PIP30           EQU     TRUE

                IF      PIP22
PATCH           EQU     110H
PATCHEND        EQU     1FFH
INPJMP          EQU     103H
INPCHR          EQU     109H
OUTJMP          EQU     106H
                ENDIF

                IF      CPM30
PATCH           EQU     188H
PATCHEND        EQU     1FFH
INPJMP          EQU     180H
OUTJMP          EQU     184H
                ENDIF

                ORG     INPJMP
                JMP     INPUT

                ORG     OUTJMP
                JMP     OUTPUT

                ORG     PATCH

DATA            EQU     84H
STAT            EQU     85H
MODE            EQU     86H
```

```
COMM            EQU     87H

INIT:           MVI     A,0CEH          ;INIT 2661 TO 9600 BAUD, 8 BITS
                STA     INITFLG
                OUT     MODE
                MVI     A,03H
                OUT     MODE
                MVI     A,27H
                OUT     COMM
                IN      DATA
                IN      DATA
                MVI     C,9
                LXI     D,SIGNON        ;PRINT THE PATCHED MESSAGE
                JMP     5

INPUT:          LDA     INITFLG         ;INPUT A CHARACTER FROM DPC-100
                ORA     A
                CZ      INIT
INPLOP:         IN      STAT
                CMA
                ANI     3
                JNZ     INPLOP
                IN      DATA
                ANI     7FH

                IF      PIP22
                STA     INPCHR
                ENDIF
                RET

OUTPUT:         MOV     A,C             ;OUTPUT A CHARACTER
                STA     SAVCHR
                LDA     INITFLG
                ORA     A
                CZ      INIT
OUTLOP:         IN      STAT
                ANI     1
                JZ      OUTLOP
                LDA     SAVCHR
                OUT     DATA
                RET

SIGNON:         DB      '2661 SERIAL PORT AT 9600 BAUD',0DH,0AH,'$'
INITFLG:        DB      0
SAVCHR:         DB      0

LAST:
                IF      LAST > PATCHEND
                ERROR PATCH IS TOO LONG !!!!!!!
                ENDIF

                END
```

(End Listings)

An Introduction to Modula 2 for Pascal Programmers

by Hugh McLarty and David W. Smith

Hugh McLarty, Logitech, 165 University Avenue, Palo Alto, CA 94301.

David W. Smith, Tolerant Systems, Inc. 81 E. Daggett Dr., San Jose, CA 95134.

If you've been following discussions about programming languages in the computer press, you have probably seen mention recently of "Modula-2, the new language by Niklaus Wirth, the father of Pascal." Articles have appeared that pit Modula-2 against Pascal and even against Ada!

What exactly is Modula-2, and how much does it differ from Pascal? How difficult would it be for an experienced Pascal programmer to learn Modula-2?

We feel that a programmer familiar with Pascal should be able to pick up the basics of Modula-2 in an afternoon and be programming comfortably within a few days. To this end, we'll briefly introduce Modula-2 from a Pascal programmer's point of view, describing similarities and differences as we go.

This will not be an introduction to the entire Modula-2 language. Parts of the language deal with direct access to the hardware and with concurrent programming. These are worth an article by themselves. The definitive description of Modula-2 may be found in Niklaus Wirth's *Programming in Modula-2*, 2nd edition (Springer-Verlag, 1983).

One warning: The following assumes some familiarity with Pascal. In particular, we will refer to what is known as Standard Pascal or ISO Pascal, which is the language standardized by the International Standards Organization (ISO), or the equivalent language standardized in the United States as ANSI/IEEE770X3. 97-1983. Most, if not all, Pascal implementations provide (incompatible) features beyond this standard, but few fail to provide the features we will mention.

The Origins of Pascal

When you hear the phrase "structured programming," chances are good that you'll think of Pascal, although you might think of PL/I, C, or even ALGOL 60 or ALGOL 68. Pascal, designed by Niklaus Wirth, appeared in the literature in about 1971. It traces its structure back to ALGOL 60, although Wirth worked on several languages between 1960 and 1971, including ALGOL-W, EULER, and PL360.

Wirth was dissatisfied with ALGOL 68 (the new improved ALGOL), at least partly because it was incomprehensible even to its designers and it did not admit of a simple, efficient implementation on typical computers. PL/I provided a similar lesson in the costs of large languages. Pascal on the other hand, was designed with simplicity and regularity as primary goals. It is characterized by features whose implementation is well understood and efficient and whose interactions are known and manageable. It is also widely associated with the now famous (or infamous) term "strongly typed language," although it was not the first language of this kind.

The Origins of Modula-2

The roots of Modula-2 are to be found in Pascal, Modula, and Mesa. Modula (i.e., Modula-1) was an intermediate stage in the evolution to Modula-2. Mesa, a language developed at the Xerox Palo Alto Research Center (PARC), was used to program the Alto and Star computers.

In 1976, Wirth spent a sabbatical year at Xerox PARC. He was impressed by Mesa and by the fact that an entire personal computer environment (operating system, compilers, editors, etc.) could be developed using a single high-level language. He returned to the Eidgenoessische Technische Hochschule (ETH) in Switzerland and designed the Lilith machine, a microcoded computer with a high-resolution bitmap display similar to the Xerox Alto. The entire software environment of the Lilith was implemented in Modula-2.

Like Pascal, Modula-2 is a relatively simple language. It contains a few simple logical control structures. It provides a number of data types, including enumerations, sets, and records, and it allows the programmer to define types. In contrast to standard Pascal, it includes a number of features that make it particularly well suited for implementing large software systems, concurrent (multiprocess or interrupt-driven) systems, and systems that require low-level access to the hardware.

Unlike Pascal, the language Modula-2 does not define a file data type or any I/O operations, such as Get, Put, Read, or Write. In a Modula-2 system, **standard modules** provide access to files and devices; these services may even be user-written. The book *Programming in Modula-2* defines several of these standard modules. This approach is familiar to C programmers, who use a standard library to do I/O.

Because of the great effect this has on Modula-2 program portability, several companies that sell Modula-2 implementations are working together to ensure compatibility of both their language implementations and their libraries.

Modula-2 and Modularity

Many dialects of Pascal allow for what is called independent compilation. The parts of a program are compiled independently and then combined with a linker or binder. While full checking is done within each compiled unit, little or no checking is done between units. For example, the number and types of procedure parameters are not checked in a call between units; in fact, it may be possible to use a variable in one unit as a procedure in another, usually with undesirable results!

The Modula-2 language stipulates that a program may consist of compilation units (modules) that are separately *but not independently* compiled. A correct implementation of the language provides as much checking between two separately compiled modules as it provides within each module. A few Pascal implementations (notably UCSD) also provide strong checking between compiled units.

When a module needs to use objects provided by another module, such as types, constants, variables, or procedures, it **imports** them from the providing module, giving the name of the providing module and the names of the desired objects. Such a module is said to be a "client" of the providing module.

Modules that provide objects for use by other modules are divided into a **definition part** and an **implementation part**. The definition part lists the objects that are provided by the module and gives enough information about them to allow the compiler to check their use. The implementation part gives the full story, including procedure bodies and other objects that are hidden within the module.

One important thing to note is that many implementations are possible for a given definition part. The details of an implementation are not visible to other modules, so any implementation that provides the objects described in the definition part and operates in the "expected" way will do.

This is called **information hiding** and it is a very powerful way of structuring systems. Some of its power comes from

the fact that most bugs, and much of what might be called software inertia, come from assumptions (i.e., information). A bug is the symptom of an incorrect assumption; inertia is the result of a widely embedded assumption that needs to be changed. Information hiding is simply the technique of localizing information; it operates on the theory that information the programmer does not have access to cannot be abused or embedded in the code. Modula-2 modules provide an excellent mechanism for doing this.

For example, during development, simple but logically correct implementations are used. As module interfaces become settled and the needs of the system become understood, more sophisticated implementations can be substituted. Altering or replacing an implementation part does not require that we understand how the rest of the system depends upon it, only that we understand how the rest of the system depends upon the objects in the definition part. If the definition part does not change, we have some confidence that no ripple effects will spread changes (and bugs) across the system.

Consider a symbol table module, as used in a compiler to collect identifiers and their meanings. We can choose an interface and write it as a definition part then create a simple implementation part, perhaps using a linear list to store symbol entries. This is slow for more than a few symbols, but it is quick and easy to implement. Later we can substitute an implementation that uses a hash table or a binary tree, or one that swaps entries to disk when memory fills up, and so on. A well-conceived definition part will remain constant through all of this, and few if any changes to client modules will be necessary.

From Pascal to Modula-2

To begin our discussion of the differences between Modula-2 and Pascal, let's consider the short programs in Figures 1a and 1b (at right). The single-entity PROGRAM of Pascal is replaced by the MODULE of Modula-2. This module is an example of a program module. Other module types are discussed below.

First, upper and lower case are distinguished in Modula-2; in particular, keywords are all upper case. "WriteLn" is distinct from "writeln" and both are distinct from "WRITELN."

The Modula-2 program imports two objects from the InOut module; in this case the objects are procedures. When this program is compiled, the definition part for InOut will be found and used to check the calls of those procedures within this client module. InOut is a standard module, described in the book *Programming in Modula-2*.

This explicit import takes a little more effort to write but quickly tells the reader what objects are being imported and where they come from. The only identifiers that are implicitly imported are record fields and enumeration constants.

Identifiers, Numbers, and Strings

The Pascal Standard says that identifiers may be of any length and that all characters are significant. The original (Jensen and Wirth) definition only required that the first eight characters be significant, and many systems still have a noticeable restriction, especially for identifiers known between modules. The Modula-2 definition does not say anything about identifier length, but it seems to be widely understood that identifiers are very long, i.e., 72 or 80 characters at a minimum. All characters are significant, and a too-long identifier is an error. We have already mentioned the new case distinctions.

Real numbers are just slightly different. No digits are required after the decimal point, but the decimal point is always required. For example:

```
PROGRAM TestProgram (OUTPUT);
BEGIN
    WriteLn('Hello, World');
END.
```

Figure 1a.
A simple Pascal program

```
MODULE TestProgram;
    FROM InOut IMPORT WriteString. WriteLn;
BEGIN
    WriteString("Hello, World");
    WriteLn
END TestProgram.
```

Figure 1b.
The same program in Modula-2

```
TYPE DIRECTIONS = SET OF (up, down, left, right);
VAR barriers: DIRECTIONS;
...
barriers := DIRECTIONS {up, left}
```

Figure 2.
Modula-2 set expression

```
RECORD x, y: REAL;
    CASE tag: Color OF                          (* first variant part   *)
      red:          a:  CARDINAL                (* note, no ( . . )     *)
    | blue:         b:  INTEGER
    | green:        c:  BITSET
    END; (* CASE *)
    z: [0 . . 99];
    CASE BOOLEAN OF                             (* second variant part*)
      TRUE:         m, n: CARDINAL
    | FALSE:        p:  POINTER TO CHAR
    END (* CASE *)
END; (* RECORD *)
```

Figure 3.
Modula-2 record expression

5. is OK in Modula-2 but not in Pascal.

1E-3 is OK in Pascal but not in Modula-2.

More importantly, Modula-2 provides syntax for hexadecimal and octal constants and for writing character constants using their octal ordinal values. Some examples:

hexadecimal:	0FFFFH	
octal:	0177564B	
character:	15C	(ASCII CR)

A character string in Modula-2 is a sequence of characters surrounded by either single or double quotation marks. A string delimited by single quotes may contain double quotes, and vice versa, but there is no way to have both in the same string.

A string of length n (n > 1) has the type:

ARRAY [0 . . n-1] OF CHAR

Presently, a string of length 1 will have type CHAR. This causes some problems, since it is not compatible with any ARRAY OF CHAR type. Wirth has agreed to relax the compatibility rules in this case, and this change should be in the next edition of *Programming in Modula-2*.

Types and Declarations

One radical change from Pascal to Modula-2 is that Modula-2 allows any number of CONST, TYPE, VAR, and PROCEDURE sections, in any order; it is also legal to make forward references within statements (e.g., to types, constants, variables, and other procedures). There is no FORWARD construct in Modula-2.

An important new scalar type, CARDINAL, takes on nonnegative integer values up to an implementation-defined maximum, usually called MaxCard. This is intended to allow use of the unsigned arithmetic that is available on many machines, and it is expected that MaxCard > MaxInt in many implementations. Many things that involve INTEGER in Pascal are done with CARDINAL in Modula-2.

Enumerations are unchanged, but subrange types are now surrounded by [and].

Set types are unchanged, although set expressions are slightly different. A standard set type, BITSET, is defined as

TYPE BITSET = SET OF [0. . n-1]

where n is the size, in bits, of a handy unit on the target computer – typically one or two words. (BITSET is part of the language, but there is no standard constant related to n!) As for set expressions, they can contain only constant elements and ranges and must be preceded by the name of a set type unless that type is BITSET (Figure 2, page 23).

Records have a slightly different syntax and are allowed to have any number of variant parts in any position (Figure 3, page 23). Modula-2 does not support "packed" structures.

Procedures

There are two minor differences between a Pascal procedure and a Modula-2 procedure. The first is that an empty formal parameter list is allowed and, indeed, is required for a function procedure with no parameters. The second is that the name of the procedure must be repeated after the terminating END of the procedure body.

However, a revolutionary change has occurred in the way procedures can be used: In addition to the procedure parameters allowed in Pascal, Modula-2 allows variables and record fields of a procedure type. Such variables and fields can have a procedure assigned to them; the procedure can then be called by designating the variable or field, followed by an actual parameter list. We have suggested some of the possibilities of this change with the fragment of Modula-2 code in Figure 4 (at left).

In a related area, Modula-2 also addresses some of the old (Level 0) Pascal problems with array parameters. In Modula-2, a procedure such as the one in Figure 5 (page 26) is possible. The ARRAY OF construction means that, when the procedure is called, an array of any size may be passed to this parameter. Within the procedure, the array has a lower bound of 0, and the built-in function HIGH returns the appropriate upper bound.

Expressions

In dealing with Modula-2 expressions, the Pascal programmer will find few surprises. The same basic set of operators is available in Modula-2, including the set operators, with the same priorities.

The one surprise in store for expression lovers is a pleasant one. Recall that the first operand of an AND or OR operator determines whether or not it is necessary to evaluate the second operand (i.e., if the first operand of an AND is false, the result is false, and the value of the second operand is unimportant). In

```
TYPE CommandProcedure = PROCEDURE(ch: CHAR);
VAR Command: ARRAY CHAR OF CommandProcedure;
        TimeToQuit: BOOLEAN;
PROCEDURE CommandInterpreter( );
VAR ch: CHAR;
BEGIN
    TimeToQuit := FALSE;
    REPEAT
        Terminal. Read(ch);                 (* get a character from key *)
        Command[ch] (ch);                   (* call command routine     *)
    UNTIL TimeToQuit;
END CommandInterpreter;
. .
PROCEDURE InitializeCommandTable;
BEGIN
    FOR ch := 0C TO 177C DO
        Command[ch] := IllegalCommand
    END;
    Command["q"] := QuitCommand;
    . . .
END InitializeCommandTable;
PROCEDURE IllegalCommand(ch: CHAR);
BEGIN
    Terminal. Write(ch); Terminal. Write("?");
    Terminal. WriteLn;
END IllegalCommand;
PROCEDURE QuitCommand(ch: CHAR);
BEGIN
    TimeToQuit := TRUE
END QuitCommand;
```

Figure 4.

Modula-2 code illustrating procedures

this case, not evaluating the second operand is called "short circuiting" the expression, and this is exactly what Pascal did *not* promise to do – but Modula-2 does.

If you don't see how important this is, consider the problem of searching a linked list for a record with a given key, where the key may not be present in the list. In Pascal, because it is not defined whether both operands of an AND are evaluated, we must write the loop very carefully to avoid accessing through a NIL pointer (Figure 6a, below). The corresponding loop in Modula-2 is more straightforward (Figure 6b, below).

Statements

One of the first things you are likely to notice when reading a Modula-2 program is the relative scarcity of the keyword BEGIN. Most statements in Modula-2 include an explicit terminator (usually END) and act on a sequence of statements, instead of on a single statement as in Pascal (although the single Pascal statement is often a sequence of statements inside of a BEGIN/END pair).

Modula-2 supports the same basic iteration and flow control structures as Pascal, with a few additions and one notable exception: there is no GOTO in Modula-2.

As shown in Figure 7 (below), the WHILE/DO is now terminated with END, and the REPEAT/UNTIL is absolutely unchanged. (In some sense, it was the most Modula-like statement in Pascal.)

To this set of iteration constructs, Modula-2 adds LOOP (Figure 8, below). Statements within a LOOP are executed until a RETURN or EXIT statement is encountered. The EXIT statement is only legal within the scope of a LOOP and terminates the nearest enclosing LOOP.

The Pascal FOR statement survives in Modula-2 with a slight increase in power. The DOWNTO form is eliminated, and a BY clause is allowed, to specify the step on each iteration. The default step is 1, of course, and the FOR statement is terminated with a matching END (Figure 9, below).

As you might expect, the IF statement requires an explicit terminating END, which is a little tedious but eliminates the problem of matching an ELSE with the correct IF clause. (The Modula-2 ELSIF construction provides some relief from the pain of IF-END nesting.) Figure 10 (page 27) demonstrates how this can remove possible problems.

The CASE statement has undergone the most change. Most notably, an ELSE clause has been added to catch missing values. This, of course, is present in one form or another in many Pascal implementations.

Case labels must still be constants, but ranges are allowed as well as single values. Also Modula-2 allows constant expressions where Pascal allows only a literal or constant identifier. The sequence of statements following a case label is separated from the next case label by a vertical bar (Figure 11, page 27).

Modules

The compilation unit in Modula-2 is a module. In general, as explained earlier, modules are split into two parts (a definition part and an implementation part), which are compiled independently. The definition part defines the interface that the module presents to the outside world: the "what" of the module. The implementation part (the "how") is intentionally hidden. To see how this works, consider a simple stack mechanism (Figure 12, page 27).

In addition to push and pop (the primary stack operators), the stack module exports two variables, Overflow and Underflow, through which errors are signaled. A module may export procedures, constants, types, and variables.

Note that the two variables declared

```
PROCEDURE Upcase(VAR str: ARRAY OF CHAR);
VAR i: CARDINAL;
BEGIN
    FOR i := 0 TO HIGH(str) DO
        str[i] := CAP(str[i])
    END
END Upcase;
```

Figure 5.
Array parameters in Modula-2

```
found := FALSE
WHILE (p <> NIL) AND NOT found DO BEGIN
    IF p^. key = k THEN
        found := TRUE
    ELSE
        p := p^. next
END;
```

Figure 6a.
Linear searching in Pascal

```
WHILE (p <> NIL) AND (p^. key <> k) DO
    p := p^. next
END;
```

Figure 6b.
The same fragment in Modula-2

```
WHILE (column < 80) AND (Card[column] = "0") DO
        Card[column] := ' ';
        column := column + 1;
END
REPEAT
        p := p^. next;
        length := length + 1;
UNTIL p = NIL
```

Figure 7.
The Modula-2 WHILE and REPEAT constructs

```
LOOP
        ReadLine( );
        IF eof EXIT;
        WriteLine( );
END
```

Figure 8.
The Modula-2 LOOP construct

```
FOR c := (Width - 1) TO Cursor BY -1 DO
        Line[c + 1] := Line[c];
END
```

Figure 9.
The Modula-2 FOR construct

in the definition part are not redeclared in the implementation part. This is because all constants, variables, and types declared in a definition part are known implicitly in the corresponding implementation part.

The third item of interest is the code at the bottom of the implementation module, where the top level of a Pascal program would be found. This code is called the module body, and it can be used to initialize the module. It is executed after the bodies of any imported modules are executed. In case of circular reference, no order is defined.

The main (top level) module of a Modula-2 program is simply a MODULE. It may (and most likely will) import definitions, but it does not export any.

How does Modula-2 ensure that the correct implementation modules are used when the program is linked? Each time a definition part is compiled, it is assigned a timestamp. When any other module is compiled that refers to that particular interface, the timestamp is included in the compiled code. Both the implementation of the interface and the clients of the interface receive these timestamps.

When modules are linked together, all of the timestamps for each interface are compared; if they are not identical, then a warning is issued. This happens if two modules are compiled with different versions of some definition part, which means that there could be an inconsistency between the definitions of objects and their use.

This section is not a comprehensive discussion of modules. We refer the interested reader to the Modula Report (in the back of *Programming in Modula-2*), particularly for further discussion of modules and low-level programming features. David L. Parnas has written several fundamental papers on the subjects of information hiding and the use of modules in the design of software systems. Doug Cooper's book, *Standard Pascal - User Reference Manual*, is a good reference for standard Pascal.

Finis

This is by no means the complete story on Modula-2. As we mentioned, we did not discuss certain provisions for concurrent programming and for low-level systems programming at all. We hope that the Pascal programmer who was curious about Modula-2 has had some questions answered and perhaps some new ones raised.

We are particularly indebted to the *DDJ* reviewers whose sharp questions and knowledgeable comments made this article much better than it would have been.

Hugh McLarty is manager of the Modula-2 Software Engineering Group at Logitech, Inc. in Palo Alto. David Smith is presently with Tolerant Systems, Inc.

■■J

```
IF Total > 2000. 0 THEN
    IF Total > 10000. 0 THEN
        Total := Total + BigBonus
ELSE
    Total := Total + SmallBonus;
```

Figure 10a.
Incorrectly indented Pascal

```
IF Total > 2000. 0 THEN
    IF Total > 10000. 0 THEN
        Total := Total + BigBonus
    ELSE
        Total := Total + SmallBonus
    END
END
```

Figure 10b.
The same rewritten in Modula-2

```
CASE ch OF
    ' ':                Token := Blank
|   '0'..'9':           Token := Number
|   'A'..'F','a'..'z':  Token := Letter
                Token := Letter
ELSE Token := Unknown
END
```

Figure 11.
A Modula-2 CASE statement

```
DEFINITION MODULE IntStack;
    EXPORT Push, Pop, Overflow, Underflow;
    VAR Overflow, Underflow : BOOLEAN;
    PROCEDURE Push (Item : INTEGER);
    PROCEDURE Pop ( ) : INTEGER;
END IntStack.
IMPLEMENTATION MODULE IntStack;
    CONST StackSize = 100;
    VAR Stk : ARRAY [1 . . STACKSIZE] OF INTEGER;
        Top : [0 . . STACKSIZE] ;
    PROCEDURE Push (Item : INTEGER);
    BEGIN
        IF Top = StackSize THEN
            Overflow := TRUE;
        ELSE
            Top := Top + 1;
            Stk[Top] = Item;
        END
    END Push;
    PROCEDURE Pop( ) : INTEGER;
    BEGIN
        IF Top = 0 THEN
            Underflow := TRUE;
            RETURN 0;
        ELSE
            Top = Top - 1;
            RETURN Stk[Top + 1] ;
    END Pop;
BEGIN (* Initialize the stack *)
    Top := 0;
    Overflow := FALSE;
    Underflow := FALSE;
END IntStack.
```

Figure 12.
Stack mechanism in Modula-2

Converting Fig-Forth to Forth-83

Forth has a colorful history compared to most modern computer languages. It evolved into its current form within a few years as the result of the work of one man (Charles Moore), was jealously guarded as a proprietary product for quite a few more years, and came into widespread use only after a group of volunteer systems programmers created a set of public domain implementations in 1978. The same programmers founded the Forth Interest Group (FIG) shortly thereafter.

Flushed by the success of the FIG implementations, our heroes formed the Forth Standards Team. The team was charged with creating a clear specification for the Forth language that could be used as a reference by both users and systems vendors. Ignoring the old engineering adage, "If it works, don't fix it," they produced the document known as the Forth-79 Standard, which was widely criticized and then almost as widely ignored.

Back to the drawing boards for the Forth Standards Team. Reconvening in late 1982, they began the traditionally grueling process of creating a new Standard, which involves much lobbying, arguing, and infighting. In August 1983, they took a final vote and approved the Forth-83 Standard document.

Even before Forth-83 hit the streets, it drew hostile comments from many vendors because it ignored or bypassed a number of important issues (such as reading device status and 32-bit implementations), was exceedingly vague about vocabularies (considered a vital feature of the language), and actually redefined the action of certain keywords and control structures that had been in common use for years – in essence creating a new language incompatible with all existing Forth programs.

The reader may well ask how such a Standards document could be created and approved. The politics of Forth would make a fascinating subject for a sociologist's dissertation, but I'm not sure I understand them well enough to explain them. For the purposes of this article, let us just note that both the board of directors of FIG and the members of the Forth Standards Team are self-elected. Nearly all have been in office since the beginning of FIG, and their responsiveness and accountability to the membership of FIG (let alone the general community of Forth vendors and users) is minimal.

With all of its peculiarities, the Forth-83 Standard seems likely to come into widespread use fairly quickly. The alternative is to stick with Forth-79, which has major deficiencies, or with fig-FORTH (Forth-78), which has become very dated. Two of the largest Forth vendors (Laboratory Microsystems and Micromotion) have already committed to converting all of their systems to Forth-83 and have independently created two rather similar implementations.

A third, public domain implementation, known as F83 and written by Michael Perry and Henry Laxen, has also become available through certain user groups. It is a powerful system that includes a full screen editor, assembler, and other programming tools. Unfortunately, F83 consists of a very large and confusing body of source code that incorporates many cute Forth programming "tricks" and shortcuts. I doubt that Forth beginners will find the F83 model comprehensible.

The remainder of this article consists of a detailed guide to conversion of existing fig-FORTH software to Forth-83. The instructions assume that the target system is a Laboratory Microsystems (LMI) or Micromotion implementation, though the instructions should be applicable for the most part to any Forth-83 system. The material assumes an average level of Forth programming competence and is not directed to readers unfamiliar with the Forth language.

by Ray Duncan

Ray Duncan, Laboratory Microsystems, P. O. Box 10430, Marina del Rey, CA 90295.

Conversion Overview

Due to the Forth Standard Team's lack of concern for upward compatibility with the commonly available fig-FORTH, poly-FORTH,™ or Forth-79 systems, it is quite likely that your programs will need some careful inspection and editing before they will execute properly on top of a Forth-83 Standard nucleus; we have provided a checklist of things to look for. Because this is not guaranteed to be inclusive, refer to the 83-Standard document and the glossary section of your Forth System User Manual for more detailed information. In case of a conflict between the two, assume the 83-Standard document description to be correct.

1. Due to the runtime requirements of the redefined **LOOP**, **+LOOP**, and **LEAVE** words, the return stack is maintained in a different format than in previous implementations. In LMI Forth systems, the **LOOP** or **+LOOP** construct requires three control words on the return stack. Programs that rely on specialized knowledge of the return stack will require extensive changes and cannot, in any case, be considered "standard" or "portable."

2. All "state smart" words such as ' (tick) and ." (dot-quote), which previously had different actions depending on whether they were invoked inside or outside a colon definition, have been either eliminated or redefined.

3. The fig-FORTH words **CFA**, **PFA**, **LFA**, and **NFA**, which were used to find the addresses of different fields within a dictionary header, have been eliminated. A new set of words, adopted from the Kim Harris experimental proposal, has been included in the new LMI and Micromotion systems:

BODY>	**LINK>**
>BODY	**>LINK**
NAME>	**N>LINK**
>NAME	**L>NAME**

Detailed explanations will come in a later section. At present, the only word of this set that is part of the 83-Standard is **>BODY**.

4. For various reasons the definition of all divide functions (except for the unsigned divide) has been changed slightly. The general effect is that quotients are floored instead of rounded toward zero. This should cause no problems for most preexisting application software. The new divide functions are marginally slower than the old (a few machine cycles under most circumstances). The side effects of redefinition for floored divide can be counter-intuitive under some circumstances. For example, in fig-FORTH the operation

```
-40 360 MOD
```

would return the obvious answer (-40) on the stack, while Forth-83 returns 320!

5. The true flag returned by all logical operations has been changed from the value 1 to the value -1 (all bits set). If your code uses 0 or 1 returned by a comparison in an arithmetic operation, you will need to interpolate the operator **ABS** after the logical operator. This is a particularly difficult problem to look for in your source code. However, I feel that this mutation in the 83-Standard was beneficial as it allows the returned true/false value to be used as a mask for **AND**.

6. **PICK** and **ROLL** are now zero-based, instead of one-based. The reasoning behind this change uses the rather weak argument that programmers frequently use zero as the beginning index for loops.

7. The word **LEAVE** has become an "Immediate" compiler word with state-checking to satisfy the 83-Standard's requirements. It, in turn, compiles the runtime word (**LEAVE**). If your previous code includes compiler extensions referencing **LEAVE**, it may need modification.

In addition, the runtime action of **LEAVE** is immediate; that is, it causes a direct transfer of control past the end of the currently active innermost loop. Consequently, the new **LEAVE** is unstructured, so that a loop increment may accidentally be passed outside the loop construct under certain circumstances. This problem is easier to demonstrate than explain; observe the following (admittedly contrived) example:

```
: TEST 1000 0 DO 2 ?TERMINAL
  IF LEAVE THEN +LOOP ;
```

If **?TERMINAL** returns a true flag, after exit from the **DO . . . +LOOP** construct, 2 will remain on the stack in the Forth-83 system whereas it would have been discarded in fig-FORTH.

8. Certain very commonly used Forth words were not included in the Standard and their eventual fate should be regarded as uncertain: they might be standardized with a different definition or abolished altogether. The list includes:

S->D	**OUT**	**ERROR**
?TERMINAL	**SCR**	**?ERROR**
CONTEXT	**DPL**	**MESSAGE**
CURRENT	**HLD**	**.LINE**
FENCE	**R#**	**THRU**
DP	**C,**	**-->**

Restrictions on an 83-Standard Forth Application Program

Forth systems, whether complying with any Standard or not, typically contain many environmentally dependent words in addition to the Standard's required word set. In order for your application programs to be portable to any Standard Forth system, you should observe the following rules (abridged from the Standard document).

1. A Standard application may reference only the definitions of the 83- Standard Required Word Set and Standard Extensions and any definitions that are subsequently defined in terms of those words.

2. A Standard program may operate only on data that was stored by the application. The initial contents of variables and arrays created at compilation time are explicitly undefined.

3. A Standard program may address:

Parameter fields of words created with **CREATE**, **VARIABLE**, and user-defined words that execute **CREATE**

Dictionary space **ALLOT**ted

Data in a valid mass storage buffer

Data area of user variables

Text Input Buffer (**TIB**) and **PAD** up to amount specified as the minimum for each area

4. A Standard program may not address:

Directly into the data or return stacks

Into a definition's name, link, or code fields

Into a definition's parameter field if its contents were not stored by the application

Although the capability of doing these types of operations is probably present in the system you are using (**SP@**, **RP@**, **>NAME**, etc.), the operations should be avoided if you intend to port your program to 83-Standard systems provided by different vendors.

Program Conversion to Forth-83

This checklist is not foolproof, but it can save you a lot of time just getting your programs to the point where they will compile (proper execution, of course, is another problem). Most of the changes indicated are related to adoption of the Forth-83 Standard; other renamings are not properly part of the Standard but have been adopted in the Laboratory Microsystems, Micromotion, or public domain Laxen/Perry models.

1. Search for all instances of **R** ; replace with **R@**.

2. Search for all instances of **-DUP** ; replace with **?DUP**.

3. Search for all instances of **WORD**. When it occurs as **WORD HERE**, delete the **HERE**. When **WORD** is not followed by **HERE**, replace it with **WORD DROP**.

4. Search for all instances of **PICK** and **ROLL** ; replace them by **1- PICK** and **1- ROLL**, respectively.

5. Examine all **DO . . . LOOP**s. For any loop that might be entered with the limit equal to the index, replace **DO** with **?DO**.

6. Search for all instances of **LEAVE**. Note that the action of **LEAVE** will be immediate. If it occurs within an **IF . . . ELSE . . . THEN** clause, **LEAVE** should be the last word before the **ELSE** or **THEN**. If **LEAVE** is used within a **DO . . . +LOOP** construct, make sure that the incrementing value will not remain on the stack if **LEAVE** is executed.

7. Search for all instances of **'** (tick) within a colon definition. If it is not preceded by **[COMPILE]** , replace it with **[']** .

8. Search for all instances of **."** (dot-quote) outside of a colon definition; replace the **."** with **.(** and the closing delimiter **"** with **)** to prevent compilation failures.

9. Search for all instances of **NFA**, **PFA**, **LFA**, and **CFA**. It is best to examine these and recode them individually, but you can make some "brute force" substitutions as follows:

This:	*Becomes:*
'	**' >BODY** (outside colon definition)
' CFA	**'** (outside colon definition)
' CFA	**[']** (inside colon definition)
NFA	**BODY> >NAME**
' NFA	**' >NAME**
LFA	**BODY> >LINK**
' LFA	**' >LINK**
PFA	**NAME> >BODY**
PFA CFA	**NAME>**

Bear in mind that the old definitions had the following actions:

CFA	(pfa --- cfa)
NFA	(pfa --- nfa)
LFA	(pfa --- lfa)
PFA	(nfa --- pfa)

These were predicated on the fact that **'** returned the parameter field address. Since **'** and **[']** now return the code field address, the new words suggested by Kim Harris revolve around that value:

>BODY	(cfa --- pfa)
>LINK	(cfa --- lfa)
>NAME	(cfa --- nfa)
BODY>	(pfa --- cfa)
LINK>	(lfa --- cfa)
NAME>	(nfa --- cfa)

These are appealing and symmetric, but before you get too carried away with them remember that a Standard program can't access any part of a dictionary definition except for the parameter field ("body"). Two additional words are provided for convenience in traversing the linked dictionary list:

N>LINK	(nfa --- lfa)
L>NAME	(lfa --- nfa)

10. Search for all instances of **ENDIF** and replace with **THEN**.

11. Search for all instances of **MINUS** or **DMINUS** and replace with **NEGATE** or **DNEGATE**, respectively.

12. Search for all instances of **SIGN** and

fix up stack logic. Usually **SIGN** can be replaced by **ROT SIGN**.

13. Any use of **-FIND** must be recoded individually. You can usually replace it with the sequence **BL WORD FIND**.

14. Search for all instances of **?** and replace with **@**.

15. Search for all instances of **BLANKS**; replace with **BLANK**.

16. Use of old **CREATE** words in your programs must be modified. In most instances on LMI implementations, you can replace it with the new word **BUILD**. The Forth-83 word **CREATE** has a much different effect.

17. Search for all instances of the construct **<BUILDS ... DOES>** and replace with **CREATE ... DOES>**.

18. Search for all instances of **END** and replace with **UNTIL**.

19. Search for all instances of **(NUMBER)** and replace with **CONVERT**.

20. Search for all instances of **IN** and replace with **>IN**.

21. Search for all instances of **FLUSH** and replace with **SAVE-BUFFERS**.

22. Search for all instances of **U*** and replace with **UM***; similarly, replace all occurrences of **U/** with **UM/MOD**.

23. Replace all instances of **S->D** with **S>D**.

24. Replace the word **SP!** with the sequence **S0 @ SP!**, and change the word **RP!** to **R0 @ RP!**.

25. Search for all instances of **TIB @** and replace with **TIB**. Recode any sequences of **TIB!** in an implementation-dependent manner.

26. Difficult to find by inspection but very dangerous is the use of the Boolean flag returned by a comparison in a calculation, such as the sequence **0= ADD**. These must be recoded individually.

27. Also difficult to find is the use of specialized knowledge of the return stack (such as fetching the index of the third outer loop). These must be individually examined and recoded.

28. Search for all **VARIABLE** declarations and delete leading initializing value (these do no harm but will be left as residual data on the stack at the end of compilation). Although most implementations do set the initial value of a variable to zero or -1, a Standard application program is, of course, not allowed to take advantage of this information. It is most correct to initialize variables at runtime (so that the code is reusable); however, your old fig-FORTH compile-time initialization of variables can easily be mimicked. For example, the fig-FORTH statement

4 VARIABLE XVAR

would be changed to

VARIABLE XVAR 4 XVAR !

29. Search for all instances of **+-** and **D+-**; replace them with **?NEGATE** and **?DNEGATE**, respectively.

30. Examine occurrences of **MOD**. If either argument can take on a negative value, the results may surprise you.

31. Find all **M/MOD** and replace with **MU/MOD**. Search for all instances of **M/**; replace with **M/MOD**.

References

Forth-83 Standard. A publication of the Forth Standards Team. Available for $15.00 from Laboratory Microsystems Inc., P. O. Box 10430, Marina del Rey, California 90295.

FORTH TOOLS, Volume 1. By Anita Anderson and Martin Tracy. A new tutorial on Forth; all example programs are written in Forth-83. Available for $20.00 from Micromotion, 12077 Wilshire Blvd. #506, Los Angeles, California 90025.

"Signed Integer Division." By Robert L. Smith. *Dr. Dobb's Journal*, September 1983, page 86.

"When a Page Is Not a Page: Forth-83 and Vocabularies." By George William Shaw II. *Dr. Dobb's Journal*, September 1983, p. 90.

■■J

(A Summary of Vocabulary Changes begins on page 32)

Summary of Vocabulary Changes in Forth-83

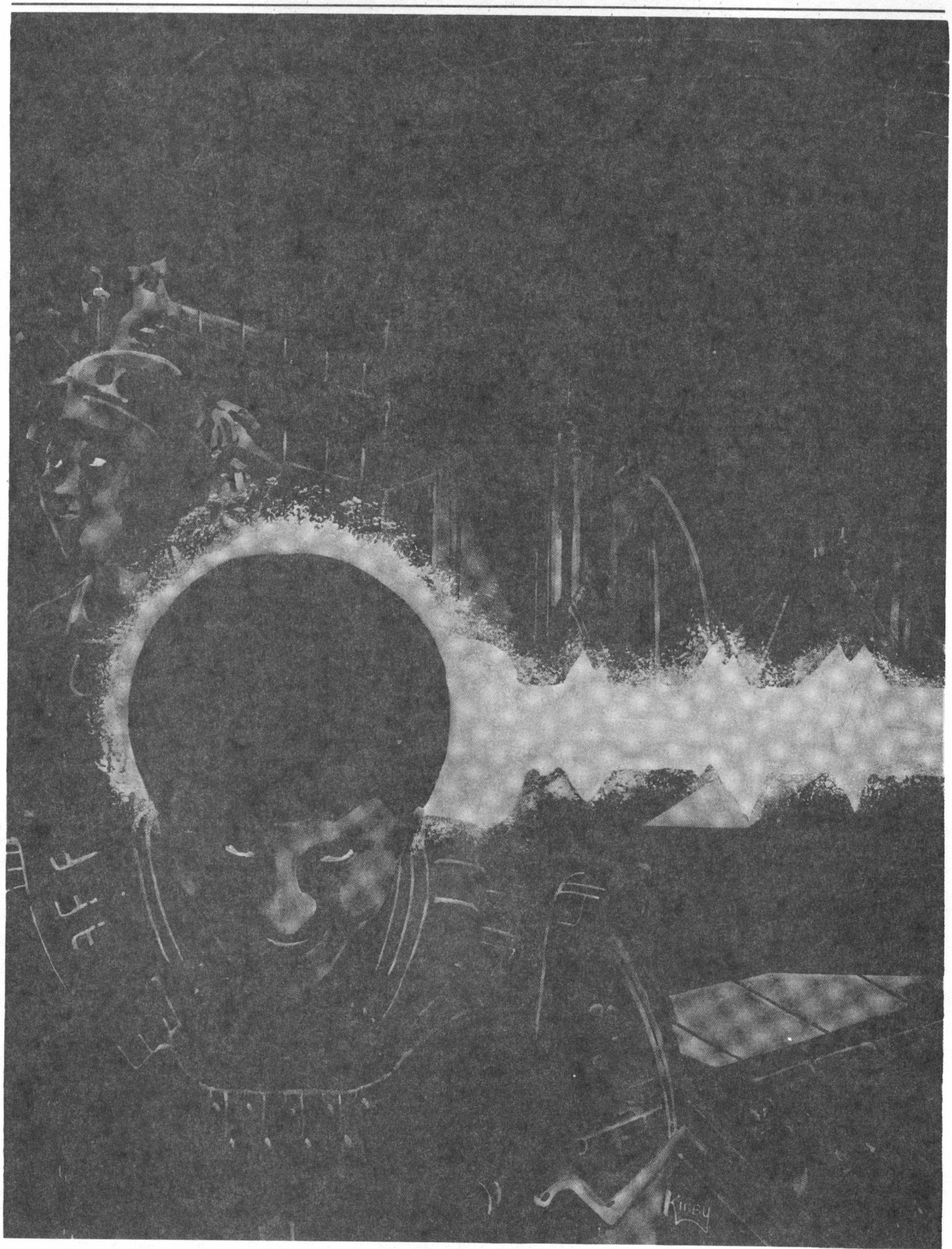
#TIB	A new 83-Standard user variable containing the length in bytes of the valid input stream within the terminal input buffer. This is *not* the length of the buffer itself. **#TIB** is set by **QUERY** after calling **EXPECT**.
	"Tick" is no longer "immediate" and returns the CFA of the target name instead of the PFA as in previous versions. Outside a colon definition, your previous code **' CFA** should be replaced by **'** while inside a colon definition it should be replaced by **['] .**
(FIND)	Deleted. To perform the function of the old **(FIND)**, use the new word **FIND**, which is 83-Standard.
+LOOP	Termination has been redefined to occur when the INDEX crosses the boundary between LIMIT-1 and LIMIT. This is implemented in LMI Forth by adjusting LIMIT to 8000H and checking for a machine overflow flag after incrementing the INDEX. This **+LOOP** is faster than in the previous version but behaves somewhat differently. For example: **1 1 DO . . . 1 +LOOP** will execute 65,536 times (in fig-FORTH or Forth-79, it would have executed only once) while **1 1 DO . . . -1 +LOOP** will execute once. See also **LOOP**.
+-	Renamed to **?NEGATE** as suggested by Laxen.
+ORIGIN	Deleted.
-DUP	Renamed to **?DUP**.
-FIND	Essentially replaced by the 83-Standard word sequence **BL WORD FIND**.
."	"Dot-quote" now may be used inside of colon definitions only. See **.(** also.
.(	New word. Immediate. The characters up to but not including the delimiting **)** are displayed on the standard output device (usually the operator's console). May be used inside or outside of a colon definition.
.NAME	Performs the function of the old **ID.**.
<BUILDS	Replaced by **CREATE**, which has some slightly different effects.
<CMOVE	Renamed to **CMOVE>** in 83-Standard Forth.
>BODY	Converts code field address to parameter field address.
>LINK	Converts code field address to link field address. Not 83-Standard.
>NAME	Converts code field address to name field address. Not 83-Standard.
?	Deleted.
?DNEGATE	New name for **D+-** as suggested by Laxen. Not 83-Standard.
?DO	Works like **DO** except executes zero times if the input INDEX and LIMIT are equal. A word suggested by Laxen/Harris/Perry. Not 83-Standard, but present in LMI and Micromotion systems.
?DUP	Same as old **-DUP**.
?NEGATE	New name for **+-** as suggested by Laxen. Not 83-Standard.
ABORT"	A new word that requires a flag on top of stack; it does nothing if the flag is false and prints a string and executes **ABORT** if the flag is true. Used like **?ERROR** but with no requirement for a disk access.
AGAIN	Still present in LMI Forth systems with function unchanged, but not present in the 83-Standard or even in the Controlled Reference Word Set. Probably fated for extinction, so its use should be avoided in new code. Can be replaced by **0 UNTIL**.
BLANK	Renamed from **BLANKS**. See below.
BLANKS	Renamed to **BLANK** (not an 83-Standard word, but included in the Controlled Reference Word Set).
BODY>	Converts parameter field address to code field address. Works like the old word **CFA**. Not 83-Standard.
CFA	Removed. See **BODY>**, **NAME>**, and **LINK>**.
CMOVE>	Previously known as **<CMOVE**.
CONVERT	Essentially works like the old fig-FORTH word **(NUMBER)**. Caution: converts positive double numbers only. Sign handling must be done outside.
CREATE	Redefined from fig-FORTH. Now used with **DOES>** in the same manner as the old **<BUILDS**.

(Continued on page 34)

D+-	Renamed to **?DNEGATE** as suggested by Laxen.
DIGIT	Returns a true flag of -1 instead of 1 as in previous version. This is not an 83-Standard word.
DMINUS	Renamed to **DNEGATE** in 83-Standard systems.
DNEGATE	83-Standard word, previously called **DMINUS**.
DOES>	Functionally similar to before, but implementation is considerably different and involves a mixture of machine code and high-level code in the defining word's parameter field.
EMPTY-BUFFERS	Marks all disk block buffers as unassigned. Does *not* write blocks marked for **UPDATE** to the disk. Not an 83-Standard word but still present in LMI Forth systems. See also **SAVE-BUFFERS** and **FLUSH**.
END	Removed. Use **UNTIL**.
ENDIF	Removed. Use **THEN**.
EXPECT	Works about the same but leaves the actual length of the input in the system variable **SPAN**.
FIND	New word that essentially provides the capabilities of the old **-FIND** and "state-smart tick."
FLUSH	Writes all **UPDATE**d blocks to disk then unassigns all block buffers. See also **EMPTY-BUFFERS**, **SAVE-BUFFERS**.
ID.	Renamed to **.NAME**. Not 83-Standard.
IN	Renamed to **>IN**.
LEAVE	Causes an immediate transfer of control to the code just beyond the next **LOOP** word. For example, in the code: **DO . . . IF XXX ELSE LEAVE YYY THEN LOOP** if the **ELSE** path is taken, the word **YYY** will never be executed (unlike fig-FORTH or Forth-79).
L>NAME	Converts link field address to name field address. Not 83-Standard.
LINK>	Converts link field address to code field address. Not 83-Standard.
LITERAL	Compilation only. The actual run-time word compiled may depend on the magnitude of the literal value.
LOAD	Loading from screen 0 is now defined to be illegal.
LOOP	Termination has been redefined to occur when the INDEX crosses the boundary between LIMIT-1 and LIMIT. This is implemented in LMI Forth by adjusting LIMIT to 8000H and checking for a machine overflow flag after incrementing the INDEX. This **LOOP** is faster than in the previous version but behaves somewhat differently. For example: **1 1 DO . . . LOOP** will execute 65,536 times (in fig-FORTH or Forth-79, it would have executed only once).
M*	No longer present in 83-Standard or Controlled Reference Word Set. Still present in LMI Forth systems, but use should probably be avoided.
M/	Renamed to **M/MOD**. This is different than the **M/MOD** that was present in fig-FORTH (which function is now carried out by **MU/MOD** in some implementations).
MINUS	Renamed to **NEGATE** in 83-Standard systems.
MOVE	This word is listed in the Uncontrolled Reference Word Set of the 83-Standard with a different action than was common in fig-FORTH. Avoid it.
N>LINK	Converts name field address to link field address. Not 83-Standard.
NAME>	Converts name field address to code field address. Not 83-Standard.
NEGATE	83-Standard word, previously named **MINUS**.
NFA	Removed. See **>NAME**.
NOT	Returns 1's complement.
PAD	Now provides a work area of at least 84 bytes.
PICK	The argument to **PICK** is now zero-based (was previously one-based). Examples: **0 PICK** is equivalent to **DUP** and **1 PICK** is equivalent to **OVER**.
R@	New name for fig-FORTH **R**.
R	Renamed to **R@** in 83-Standard.
ROLL	The argument to **ROLL** is now zero-based, instead of one-based. Examples: **0 ROLL** is a null operation, **1 ROLL** is equivalent to **SWAP**, **2 ROLL** is equivalent to **ROT**, etc.
RP!	Takes its argument from the top of the data stack rather than **R0**. A change suggested by Laxen/Perry for symmetry with **RP@**. The word **RP!**

	in your previous code should be replaced with the sequence **R0 @ RP!**. Not 83-Standard.
S->D	Renamed to **S>D** as suggested by Laxen. Not 83-Standard.
SAVE-BUFFERS	All buffers marked as **UPDATE**d are written to the disk but remain assigned. See also **EMPTY-BUFFERS** and **FLUSH**.
SIGN	Definition changed: takes its argument from the top of stack rather than the third item on the stack. The word **SIGN** in your old code can be replaced by **ROT SIGN**.
SP!	Takes its argument from the top of stack rather than **S0**. A change suggested by Laxen/Harris/Perry for symmetry with **SP@**. The word **SP!** in your previous code should be replaced with **S0 @ SP!**. Not 83-Standard.
SPAN	A new system variable that contains the length of the last input via **EXPECT**.
STATE	System variable. In LMI Forth systems, the value of **STATE** is -1 if compiling, 0 otherwise. An 83-Standard application shouldn't modify this variable.
TIB	Definition changed. Formerly returned the address of the location containing the address of the terminal input buffer. Now returns the actual address of the buffer. In your existing code, replace all sequences **TIB @** by the word **TIB** alone.
U*	Renamed to **UM***.
U/	Renamed to **UM/MOD**.
UM*	New name for **U***.
UM/MOD	New name for **U/**.
VARIABLE	Does not accept an initializing value.
VLIST	Renamed to **WORDS** (suggestion of Laxen).
VOCABULARY	The 83-Standard, in an attempt to remedy the 79-Standard's restrictiveness on vocabulary structures, left most details to the imagination of the Forth implementor. A radically different experimental proposal was offered by Bill Ragsdale and printed as an appendix to the Standards document, but this proposal was not approved as part of the Standard proper and has not been adopted by all Forth vendors at this point. Beware!
VOCS	(LMI implementations) Displays the current search order and the names of all declared vocabularies in the system. Not 83-Standard.
WIDTH	No longer present in LMI implementations. The full name (up to 31 characters) is always used.
WORD	Defined slightly changed from fig-FORTH. Returns the address of the first byte of the token it scanned off the input stream. A token may be up to 255 bytes in length.
WORDS	Displays the names of the definitions in the vocabulary that is first in the current search order. Similar to old **VLIST** command. Not 83-Standard.
[']	Used inside a colon definition to compile the CFA of the following word, like the old "state-smart tick" word. See also **'** and **FIND**.

(End Table)



—Doug Kirby

Sixth Generation Computers

Dr. Dobb's *periodically prints material designed to provoke thought on topics which look toward the future. In December 1982, we published an essay by Richard Grigonis entitled "Fifth-Generation Computers," which sketched a picture of microcomputer technology in the 1990s. Michael Doherty challenged his predictions (Open Forum, March 1983), requesting more substantial discussion. Grigonis' lengthy response in the August 1983* DDJ *discussed topics ranging from silicon theory to sociology. Now Grigonis has provided us with more food for thought. Doherty, having seen an advance copy of the piece, has already written a reply, which we will publish next month. – Editor*

The favorable response to my last article, "And Still More Fifth Generation Computers" (*DDJ* No. 82), has encouraged me to write another article on future hardware that picks up where the last one left off, taking speculation about future computers to the limits that can be conceived of by the human mind. So, without further ado, let us complete our journey into the future....

Physical Limitations

All artificial intelligence programs must represent knowledge and the logical connections between such knowledge, the total possible combinations of these being the search space or all possible final states or solutions. A tree structure is usually the most convenient way of describing the possible final states, with each successor node representing a possible state that can be derived from the initial state via the appropriate operators.

The search for a desired goal, such as the best move in a chess game, thus consists of searching each successor node of the tree (graph) to discover a node with a state description satisfying the goal; the path to that node is the appropriate operator sequence that yields the best final state.

Blind search or "exhaustive search" algorithms (such as the breadth-first search) never fail to uncover the best solution since they examine every node or possible state. Unfortunately, many problems have a search space that is either infinite (such as logic) or else so large (such as chess) that the computational cost makes such exhaustive searches impossible. The fact that one must examine all sequences of n moves immediately implies a search space where the number of nodes grows exponentially with n, so if at each move the player has K choices, then n moves comprise K^n possible move sequences. The possible final states grow exponentially with n and so result in a combinatorial explosion that overflows the computer's memory and requires eons of processing time. The algorithm for playing a perfect game of chess is not beyond the capabilities of a theoretical universal Turing machine, but it is certainly beyond its physical embodiment, the digital computer.

Developing artificial intelligence thus became a matter of devising short-cuts, or forms of heuristic search, so that information about the nature of the problem domain allows the program to "prune the search tree" and run more efficiently. Some of these heuristic searches include the Minimax and Alpha-Beta pruning procedures. Still, the danger with pruning the search tree via short-cuts (heuristics) is that, while some of these heuristics do guarantee success by improving efficiency, others do not even guarantee a solution at all! This is particularly troublesome when one is dealing with yes-no questions, as opposed to those questions requiring a value (with a tolerable margin of error) as an answer. The greater the number of heuristics, the more like a human being the program will behave, and the greater the chance that the program will miss the best solution. Human beings use heuristics a good deal more than they do precise algorithmic techniques, and the certainty of their decisions suffers accordingly. Only an exhaustive search of the available possibilities can give a perfect solution, but the computational cost of performing these searches is tremendous.

Therefore, many mathematical, logical, and artificial intelligence problems can never be solved because the computational requirements of algorithms that perform better than the heuristic capabilities of humans exceed the ability of any existing or projected computer. Even an optical computer utilizing matrix processing techniques will probably never exceed 1.5 trillion operations per second; this signal flow falls far short of even Hans J. Bremermann's relatively small theoretical limit of 1.35×10^{47} bits per second per gram of system weight. Searching through a search space requires computations, and computations require time and energy. Since computational devices are bound by both limits here in the physical world, as Bremermann says, "the accessible portion of the mathematical universe is limited."

Or is it? Memory is not the problem it used to be. The new Cray and Fujitsu supercomputers can each be equipped with 256 megabytes of internal memory, and the Japanese Fifth Generation computer will have 1000 megabytes or about a billion bytes (gigabyte) of memory. Sometime in the first half of the next century, supercomputers will have at their disposal about 64 gigabytes of memory. It is the *processing time* required for a solution to many of these algorithms that seems to be the real problem. In the case of chess alone, with over 10^{120} moves (I think Shannon's estimate of this is wrong, as the real figure probably exceeds 10^{150} moves), a huge number of computation-years are required to exhaustively search the possible moves.

The two solutions to this problem have been either to develop distributed array processing (reducing a problem to its components and having a different processor work on each part of the problem) or to simply build faster processors with smaller or newer non-von Neumann architectures. Neither of these ideas has turned out to be quite as successful as was initially thought. After all, even if every atom in our galaxy were replaced with a Cray-1 supercomputer, it would still take centuries to search through all the possible chess move sequences! Data flow devices have not demonstrated very many advantages over conventional computers (indeed it is impossible for them to implement simple loops or recursion), and faster and faster processors designed along the von Neumann architecture (even ignoring the problems posed by the so-called "von Neumann bottleneck") must be constructed smaller and smaller until they literally disappear.

We must conclude then that even the much-touted Japanese fifth generation computer is doomed to fail in the artificial intelligence and predicate logic theorem resolution tasks envisioned by its sponsors.

by Richard Grigonis

Richard Grigonis, Children's Television Workshop, One Lincoln Plaza, New York, NY 10023.

All drawings by Doug Kirby.

Superluminal Processors?

A novel solution, which everyone seems to have missed, is to devise a processor whose signals are not bound by the speed barrier imposed by the special theory of relativity – in other words, to build a computer where the processor signals can travel faster than light. This sounds impossible, but in fact we may achieve this in three ways:

(1) Tachyons
(2) The application of the Einstein-Podolsky-Rosen (EPR) effect
(3) The so-called "advanced potentials" solution of the moving charge equations derived from Maxwell's electromagnetic theory

Let us now examine each of these three possibilities in detail and see what we can come up with.

Tachyons

Although ordinary particles with finite, real rest masses and energies cannot travel faster than electromagnetic radiation with respect to any frame of reference, particles having imaginary quantities of energy and momentum can indeed travel faster than light. These hypothetical particles were named *tachyons*, from the Greek *tachys*, meaning "swift."

In 1962 three American physicists O. Bilanuik, V. Deshpande, and E. C. G. Sudarshan at the University of Rochester mathematically demonstrated that tachyons could have *real* energy and momentum if the particles possessed an *imaginary* rest mass instead of a real one. The value for the rest mass is imaginary in that it is the square root of a negative number.

But another problem was related to the faster-than-light (or "superluminal") velocities of such hypothetical particles. Since anything traveling faster than light should, from our frame of reference, travel backwards in time as well, an observer watching a tachyon being exchanged between two atoms (say, between atom A and B) would see a tachyon particle of *negative* energy absorbed by atom B *before* it is emitted by A. Two things are wrong here. First, particles of negative energy are not allowed; second, the tachyon violates causality by traveling faster than light and so appears to reach atom B before it is emitted by atom A!

These theoretical obstacles to the existence of tachyons were eliminated by Columbia University physicist Gerald Feinberg and the "reinterpretation principle" he devised in 1967. An observer in the correct frame of reference (a high relativistic velocity) would not see a negative-energy tachyon being absorbed by atom B before it is emitted by A. Since emitting a negative amount of energy is equivalent to absorbing a positive amount, the observer instead would see *atom B emit a positive-energy tachyon*, which is then absorbed by atom A. The problems of both negative energy and time-reversal vanish.

Because they have an imaginary rest mass, as tachyons lose energy they speed up instead of slowing down! Tachyons therefore would be difficult to control. Whereas an ordinary particle is stationary when at rest and requires the application of an infinite amount of vector forces (energy) to attain the speed of light, a tachyon is subject to the opposite principle: it travels at infinite velocity when at rest, and subjecting it to an infinite amount of propulsive force would merely slow it down to the speed of light!

The speed of light, then, is a "barrier" between our universe and the universe of tachyons. Interestingly, the time-axis of the tachyons' world lines (as described by a classical Minkowski space-time diagram) is *perpendicular* to the time-axis of the particles in our own universe, so that once an observer arrives in a tachyon universe, the tachyons would appear to behave like normal particles – but the particles in our universe, by comparison, would now appear to be tachyons! That, of course, is definitely another story.

In any case, for a tachyonic computer to function, it is necessary for the tachyon signals to interact with ordinary matter at some point – namely, by triggering the processor's binary switches.

If we assume that a tachyon can carry an electrical charge, we can create tachyons by bombarding atoms with high-energy photons in the form of, say, gamma rays. This then liberates pairs of tachyons, the resulting particles having equal and opposite charges. This electrical charge on the tachyon, because it is traveling at superluminal speed, emits a special form of electromagnetic radiation called Cerenkov radiation. This type of radiation, first fully described by the Russian physicist Pavel A. Cerenkov in 1937, is the cone-shaped shock wave of photons emitted by a particle traveling faster than light in a medium (not a vacuum, obviously, since ordinary particles do not travel faster than light); it is sort of a light version of the "sonic boom." (Cerenkov radiation, when generated by particles whirring around in a cyclotron, looks bluish white, but, strangely enough, a prism does not separate this light into a spectrum.) A tachyon accelerates to infinite velocity within a distance of only 0.001 centimeter, as the particle quickly divests itself of its electrical charge by liberating a Cerenkov radiation cone with a 90-degree opening.

A distance of 0.001 centimeter is too small for the construction of a super-

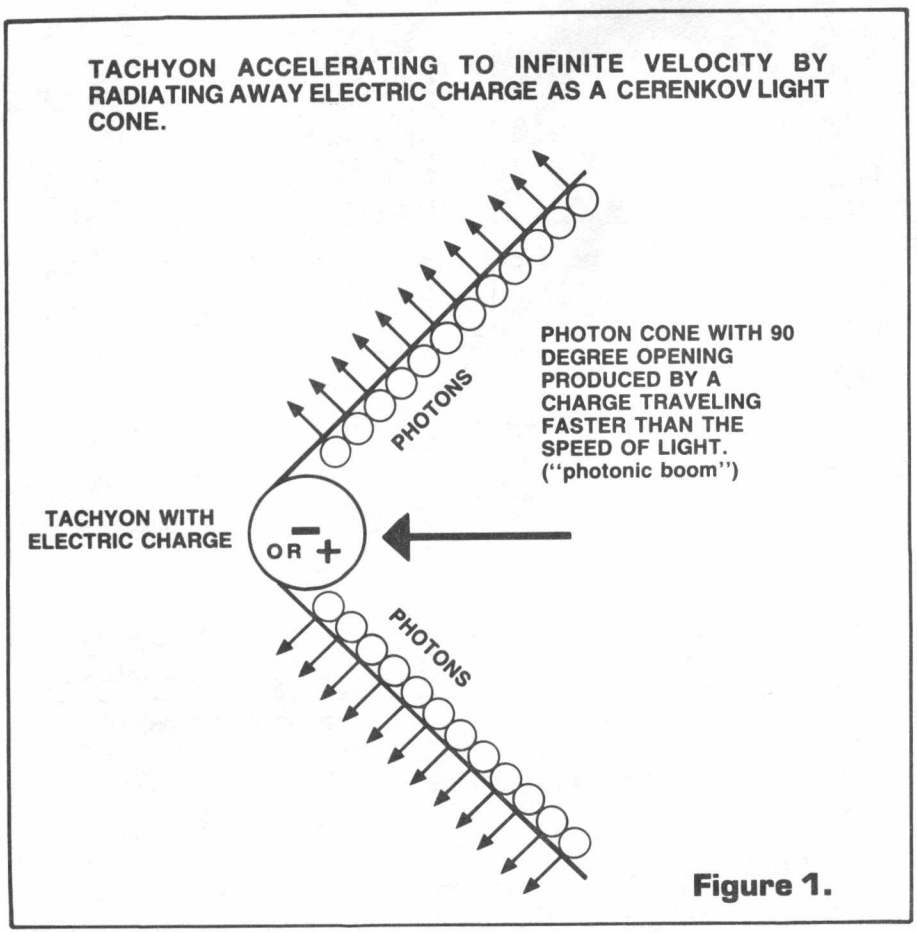


Figure 1.

luminal processor switch, but a solution exists. The tachyons can be "refreshed," or given an electrical charge again, by placing an electric field in their trajectory. Once recharged in this way, the tachyons again divest themselves of their newly acquired energy via Cerenkov radiation, which can now be detected by a photomultiplier tube.

Therefore, each switch of a tachyonic computer would consist of a powerful electromagnetic source to generate the tachyons and an electrical field/photomultiplier tube combination to detect tachyonic signals coming from other switches. Since we have no way of transmitting tachyons in a particular direction, all of the switches would have to broadcast and receive signals only at certain timed intervals – in a sort of packet-switching arrangement.

Another problem is that an experiment by T. Alvager and M. N. Kriesler at Princeton in 1968, using a gamma ray source (radioactive cesium) and an electric field/light detector apparatus, did *not* detect the transmission of any tachyons! However, if high-energy tachyons merely decay into several faster and more stable tachyons, this would have accounted for the Princeton experimental results.

If tachyons are passed from one atom nucleus to another in straight lines, then disturbing one end of an object would cause a tachyonic pulse to pop out of the other end instantly. We may have here the solution to the directional transmission problem. Perhaps future computers will be partly mechanical, with the switches disturbing in some way the ends of rods, each rod composed of a highly compressed substance like metallic hydrogen (although, come to think of it, *any* substance will turn into a metal if subjected to sufficient pressure); the computer would transmit faster-than-light tachyonic pulses through a network of these rods.

Ironically, in the 1940s and 1950s, certain parts of computer processors ran so quickly in comparison to the rest of the equipment that the signals had to be slowed down at certain points by converting the signals from electrical pulses moving through wires to waves moving through troughs filled with a liquid. The Americans used mercury for this purpose, while Alan Turing thought that gin had the right density.

The EPR Effect

Einstein never liked the indeterministic, probabilistic view of the world posited by quantum mechanics. Einstein's own theories of relativity – the other half of modern physics – are actually the last of the great classical theories; their equations specify that space, time, and mass are a precise, calculable function of velocity,

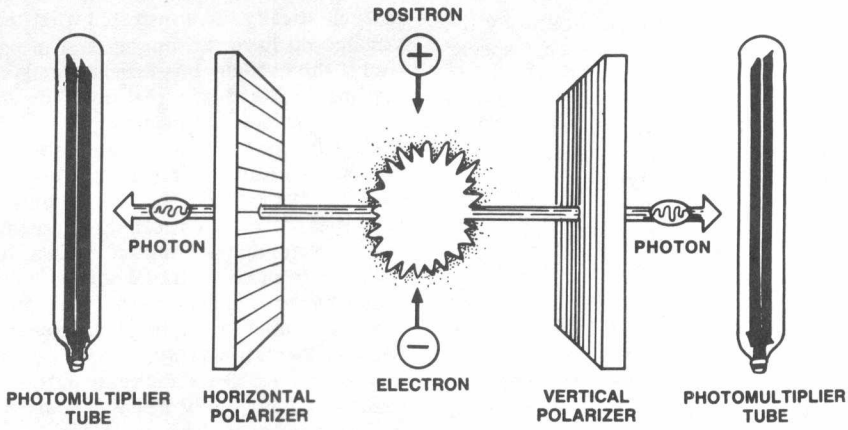


Figure 2.

EXPERIMENTAL APPARATUS TO DETECT EPR EFFECT BETWEEN TWO PHOTONS.

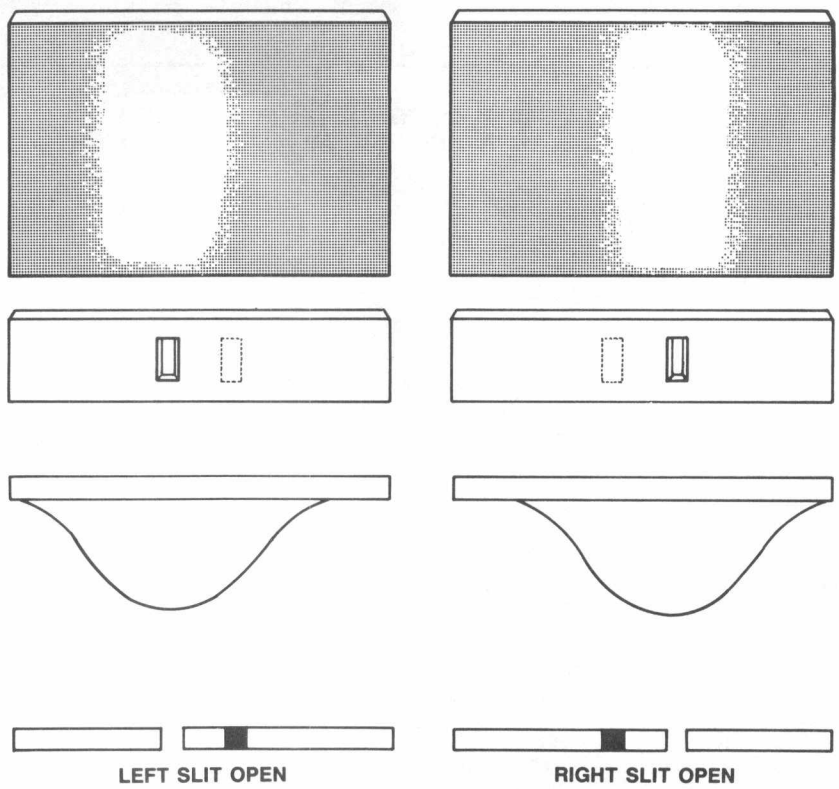


Figure 3.

and that the effect of being at rest in a gravitational field is the equivalent to being at rest in an accelerated coordinate system. Quantum physics, on the other hand, is dominated by two things that disturbed Einstein: the Schrödinger wave equation and the Heisenberg uncertainty principle.

To illustrate these two bizarre items, we reintroduce an old high school physics experiment. Most of us may remember that in 1803 Thomas Young demonstrated the wave nature of light by directing sunlight through a card or metal plate with two vertical slits in it. The diffracted light, when projected upon a wall or screen, yields an interference pattern with alternating bands of light and darkness, the brightest band at the center. With only one slit open, however, diffraction still takes place, but the light forms a single fuzzy vertical patch on the screen, corresponding to the single slit through which it has just passed.

Since not only photons but all subatomic particles have a wave aspect as well as a particle aspect, we can run the double slit experiment again, this time using electrons. Instead of an ordinary screen, however, we need one coated with a phosphor to make the electron impacts visible as tiny flashes of light (turning the apparatus into a sort of CRT); a cloud chamber would also suffice.

When a single photon is diffracted through one slit, we cannot determine exactly where on the screen the particle will land. According to the Heisenberg uncertainty principle, we can know either the momentum of a particle with accuracy or the particle's position with accuracy, but we can never determine both values with high accuracy! Since we know the momentum of a photon (traveling at light velocity), we therefore cannot determine its position – until it hits the screen.

The Schrödinger Wave Equation

But if we cannot determine the photon's precise position until it hits the screen, then can we at least determine the chances of it hitting any particular area of the screen? Yes. This is where the Schrödinger wave equation comes into play, which is simply a partial differential equation that allows one to compute, given the initial state of a system, all of the probable final states of the system. In our case, this would be all of the possible positions on the screen the photon could hit at any particular time. Instead of regarding the present state of the universe as being determined by the past and in turn determining the future (the classical view), the Schrödinger wave equation gives us a "wave function" of many possible future and past states. Thus, the future state of a physical system (until the particle finally hits) is a superposition of *all* possible outcomes.

An infinite number of states of varying probability can represent possible solutions to the Schrödinger wave equation for any physical system, depending upon the measurement. Each state is a solution to the wave equation and a possible future state (possible impact point). So, the particle can be anywhere in the cosmos before it is observed by its impact! After all, the photon or electron may have missed the screen entirely and is now far away, but this is a small probability. It is important to note that the system is not in *one* of these states, but it is probabilistically in *all* of the states until the measurement occurs; the representation of the "ensemble" of states is known as the *state vector*, or the "statistical ensemble" of states.

So imagine an electron or photon diffracted through a single slit, and imagine its wave function spread out over a huge area that includes the screen. What is so irritating about all this is that, for all practical purposes and indeed for *all* purposes, the "particle" does not exist as such until it is measured by an observer. At the moment of measurement – which is the moment when the particle hits the screen – all of the possible states disappear, the system undergoes a discontinuous change, and we find the system to be in one particular state, one given by the position on the screen of the particle's impact. The act of measurement is thus said to "collapse" the state vector onto one of the possible states given by the Schrödinger wave equation. This is also called the "reduction of the wave packet." Strangely, the particular state (position on the screen) the system settles on is determined only by pure chance.

So, although the particle is probabilistically "everywhere" (to the point where if both slits are open it can appear to pass through both slits simultaneously and interfere with itself like a wave), it is also "nowhere" in that it doesn't exist until we measure it! What is even more disturbing is that, since the wave function covers a huge area that only includes the screen, we suddenly realize that the collapse of the state vector occurs instantaneously, or much faster than the maximum speed (light velocity) allowed for by the special theory of relativity! Things like this used to drive Einstein right up the wall, leading him to say at one point that "God does not play dice with the universe."

Einstein's "Paradox" Isn't

Because Einstein could not find any inconsistency in quantum mechanics, he decided instead to prove that quantum theory was an incomplete theory. He thought he had achieved this after he cleverly concocted a "thought experiment" with two of his colleagues, Boris Podolsky and Nathan Rosen, in 1935. Unfortunately for Einstein, this particular "thought experiment" ultimately backfired, proving him wrong and quantum mechanics right. The idea is now known as the EPR (for Einstein, Podolsky, and Rosen) effect, or the EPR paradox.

There are many ways of illustrating the paradox. For example, let's bump two electrons together and create a two-particle system of zero spin, which means that the spin of one electron always cancels out the spin of the other. For the moment we will not measure the electrons to determine which way they are spinning. Next, let us separate the electrons by several light years, leaving one of them in our laboratory. After doing this we measure the spin of "our" electron, thus collapsing the state vector. Theoretically, the other electron doesn't exist until we measure it too. But we already know that both electrons make up a single two-particle system of zero spin, and the total spin is *always* going to be zero. So, even though the other electron is light years away, we already know what the value of its spin is by measuring the spin of the electron here on earth! The situation becomes even more bizarre when we remember that the other electron's spin is always going to be opposite to the spin of the one in the lab, so if we could alter the spin of the electron here on earth we would also *instantaneously* change the spin of the faraway particle!

Kind of scary, eh? Well, it's only scary if an experiment demonstrates it to be true. Einstein's argument was that such a thing would never happen, because a light signal could not travel fast enough to connect the two particles. Einstein reasoned that quantum theory must be incomplete because it allows such faster-than-light (superluminal) connections between two particles, a phenomenon that must be impossible, violating as it does Einstein's own special theory of relativity.

Einstein believed in the principle of local causes – what we measure or do to one electron's spin here on earth (let us call it position S_1) cannot possibly affect the other electron's spin in deep space (let us call it position S_2) unless we allow sufficient time for some kind of signal traveling at the speed of light to reach it and impart the information to the particle. As Einstein wrote: "One can escape from this conclusion [of the incompleteness of quantum mechanics] only by either assuming that the measurement of S_1 'telepathically' changes the real situation of S_2 or by denying independent real situations as such to things which are spatially separated from each other. Both alternatives appear to me entirely unacceptable."

Amazingly, Einstein and his col-

leagues were proved wrong! *There are no local causes.*

Bell's Theorem

In 1964, a physicist named J. S. Bell at the European Organization for Nuclear Research (CERN) in Switzerland developed a mathematical proof that was subsequently strengthened by others and is now known as Bell's Theorem. With it, Bell theoretically demolished the principle of local causes. In 1969, J. F. Clauser, M. A. Horner, Abner Shimony, and R. A. Holt showed how physical experiments could be performed to test for faster-than-light (or "superluminal") connections or "correlations" between particles. The first of these tests was made in 1972 by Clauser with S. J. Freedman.

The scientists electrically excited neon atoms so that, when the atoms' electrons fell back to a lower, more stable energy state, they emitted pairs of photons moving in opposite directions. The wave motion of photons can be plane polarized (vertical or horizontal) or circularly polarized (clockwise or counterclockwise), so that the plane of the wave displacement rotates as they move through space. Since these photons have a common origin, they can be considered as a two-particle system. If one of the two photons is vertically polarized, the other one is also vertically polarized. If one photon is horizontally polarized, then so is the other one. Therefore, if we measure the polarization state of one photon, we also know the value of the polarization state of the other photon, even though we have not measured it and it is far enough away so that the "information" must be traveling faster than light.

The Clauser-Freedman experiment did just that by placing polarized filters on either side of the light source. Photons passing through these polarizers then encountered photomultiplier tubes. When both polarizers were vertically oriented, the vertically vibrating photons passed through and set off both photomultipliers simultaneously. When one of the polarizers was rotated at 90 degrees to the other, however, one set of photons was stopped and the photomultipliers did not fire together.

Another way of generating pairs of photons is to bring an electron and a positron (antimatter electron) into contact, causing their annihilation. In the case of photons produced by matter-antimatter annihilation, quantum mechanics predicts that the two photons will have opposite polarization. In this case, then, the photomultipliers will fire together only if both polarizing sheets are set 90 degrees to each other. And it works again!

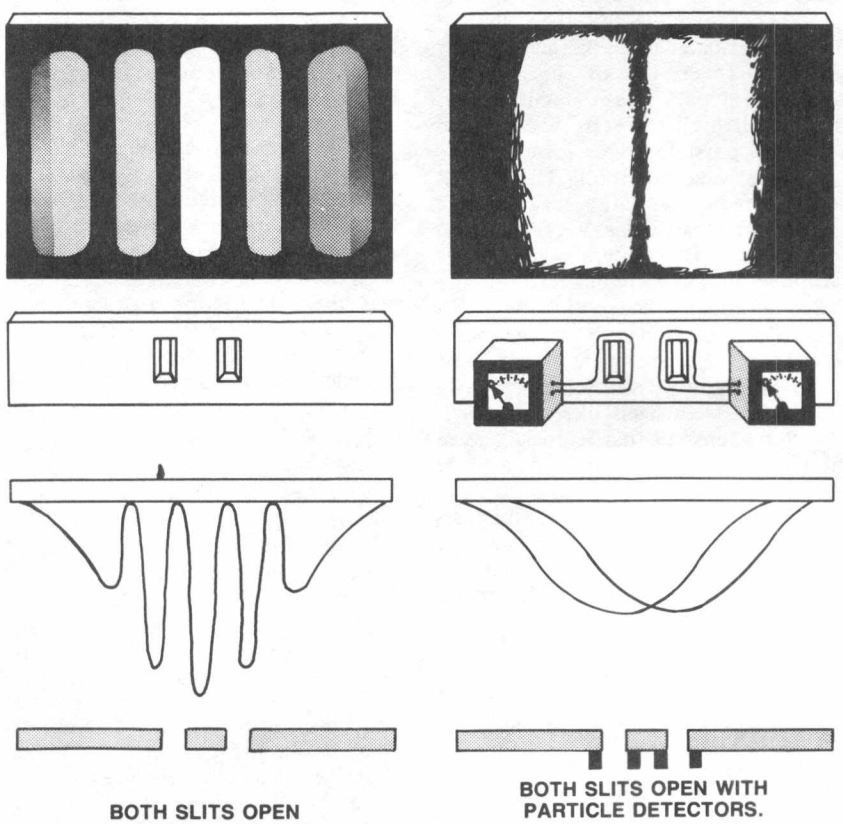
The most recent experiment along these lines (the eighth since 1972, I believe) was performed by Alain Aspect, Philippe Grangier, and Gerard Roger at the University of Paris. These experiments seem to prove once and for all that there are no local causes. This implies that a system can never be described in isolation, since all of the particles of matter in the universe interacted with each other long ago during the Big Bang and must therefore be "correlated" or connected.

Transmission Problems

Does all of this mean that we can build optical computers that operate faster than the speed of light? By changing an individual electron's spin or a photon's polarization, it would be quite simple to transmit messages faster than light. Quantum mechanics, however, discourages manipulating individual particles once they are measured; this violates the essential indeterminism upon which quantum theory is based. Nevertheless, while it is true that an individual photon's polarization cannot be altered once it is measured, it is also true that we decide what we are going to be looking for in the first place. Although quantum mechanics forbids individual particles to be causally tampered with, we could manipulate the variable values of a statistically large number of such particles, monitored at whatever remote locations we wish. What immediately comes to mind is separating and altering the spin of a stream of electrons with Stern-Gerlach devices, as suggested by Gary Zukav.

Other Superluminal Phenomena

Actually, we have already encountered many kinds of faster-than-light phenomena in the double-slit experiment. First, by simply measuring the system, we have seen the wavc function instantly collapse at faster than tne speed of light. However, another strange phenomenon occurs that the average high school student misses when working with the apparatus. The moment an observer covers up one of the slits, preventing one set of photons or electrons from passing through it, the interference pattern consisting of light and dark bands disappears and is replaced by the single fuzzy band indicative of photons or electrons diffracted through a single slit. Now, at the moment that the observer covers up the slit, how do the other particles that are passing through the other slit "know" that there



DOUBLE-SLIT EXPERIMENT(cont.)

Figure 4.

is now only one slit and that it is now permissible to go into what would have been the dark bands on the screen were both slits open, thus giving us the single fuzzy band characteristic of a single slit?

The problem is accentuated further when we fire one photon or electron at a time through the slits, slowly generating the single fuzzy band or the interference pattern on the screen, depending upon whether one or two slits are open. In this case the photon or electron not only knows what the other particles are doing, but it also knows what the physical configuration of the apparatus is! How does a single particle know if one or two slits are open?

Indeed, things become even more disturbing when one realizes that, since the spacing between the light and dark bands on the screen is determined by the distance between the slits, the particle's behavior also depends upon how far it was from a slit it didn't go through! Does the particle actually (because of its probabilistic wavelike properties) "divide" into two parts and "wave" through the two slits, interfering with itself?

To find out, let's run the experiment again with electrons, but this time try to outwit each particle and pin down exactly which slit (or slits) it goes through by building a particle detector. This is done by placing a loop of wire around each slit. A particle in motion carrying an electrical charge (the electron) generates a magnetic field; when these lines of force cut through the loop of wire (through a photon exchange between electron and wire), it generates an electrical pulse as the particle passes through the slit. We can record this pulse. By firing one electron at a time at the apparatus, we find that each time an electron hits the phosphor-coated screen, only one particle detector will register – never both. By measuring the electron at the slit we have eliminated its probabilistic location in space and have collapsed the state vector to a particular position, which is the particular slit it has passed through. But there is more! With both slits open in this experiment, the interference pattern of multiple bands has vanished. Instead we find a pattern of *two* vertical fuzzy bands, side by side, each one resembling the pattern produced by a single slit.

What happened? Consider the uncertainty principle. By measuring the position of the electron immediately (at the slit), we force the particle to interact with the apparatus by sending a photon (the exchange particle of electromagnetism) to the loop of wire. But when a photon interacts with an electron, its momentum (mass times velocity, Planck's constant divided by the wavelength) changes by an unknown amount. This is the essence of the uncertainty principle and the price we pay for knowing the electron's position. Even worse, the value for the momentum of each electron is totally different from that of any other electron, as their positions are detected in turn.

Because we have destroyed the electron's probabilistic location in space by measuring it at a particular position (one slit or the other), the electron no longer behaves as a wave and so cannot "wave" through both slits. Instead it behaves more like a particle, the interference pattern of many light and dark bands disappears, and we see the patterns of two "single" slits side-by-side on the screen. When we do not know which slit each electron passes through, the interference pattern reappears! Sneaky particles, eh?

Interestingly, if we use an "unreliable" detector, one that merely tells us that there is a probability of, say, 0.7 that an electron went through a particular slit, we will find that the interference pattern only partly disappears – the multiple bands are made a bit fuzzy – and upon this is superimposed, also a bit fuzzy, the two-single-slit pattern!

TRANSMISSION AND ABSORPTION OF SPHERICAL ELECTROMAGNETIC WAVES FROM A HERTZ DIPOLE EARTHED ON ONE SIDE.

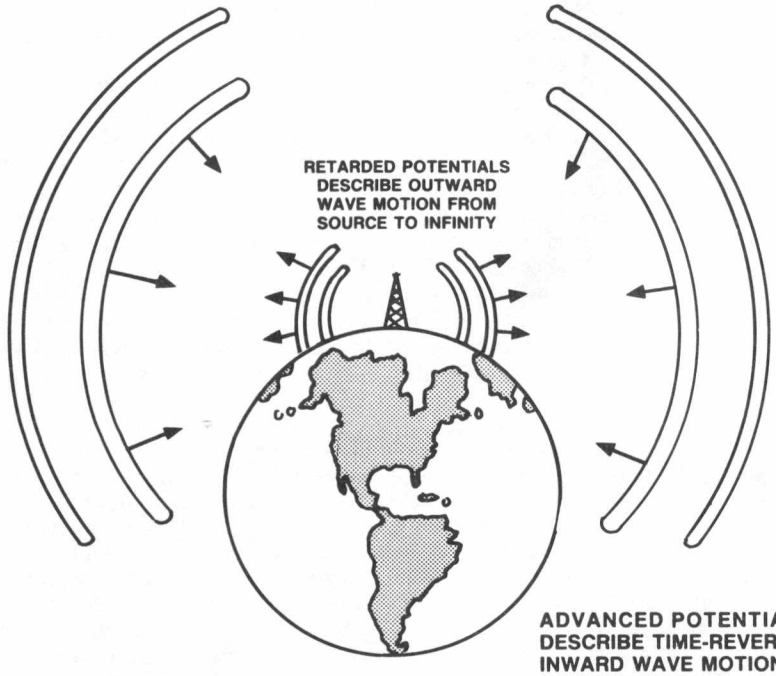


Figure 5.

Backward Causation?

One could resolve all of these strange phenomena (and eliminate the idea of an actual signal traveling faster than light) by assuming that the "direction of causality" might be violated when we measure something. In other words, instead of regarding the initial state of the system as determining the outcome of measurements made on it, we might regard the outcome of the measurements as determining the initial state!

As long ago as 1947, Costa de Beauregard thought that the quantum information about electron spin or photon polarization (of the EPR paradox) travels backward in time from the measured particle to the event that produced the two particles, then forward in time to the other particle. The information thus appears at the second particle just as we measure it at the first particle!

More recently, John A. Wheeler, the brilliant physicist who coined the term "black hole" some years ago, has devised a thought experiment (called the "delayed-choice, double-slit experiment") that suggests that quantum mechanics seems to allow – indeed to demand – that backward causation is real. Essentially, it consists of the same version of the experiment that used the "electron detector," except this time we are working with photons. A timed photon goes through the plate with the two slits. Beyond the two slits are two different types of optical measuring equipment.

One piece of equipment can measure

the photon as a *wave*. It doesn't determine which slit the photon goes through, thereby allowing the photon to move probabilistically as a wave through both slits and to interfere with itself; this "causes" the photon to strike only those areas of the screen corresponding to the bright bands of the interference pattern (areas of constructive interference), thus triggering this wave detector. The other device can measure the photon as a *particle*, by detecting through which slit the photon passes.

Nothing appears new in this experiment thus far; it just sounds like the previous experiment with the electrons and the loops of wire. But before continuing, it should be noted that, although photons (and other particles) appear to have the properties of both particles and waves, they never display both properties simultaneously. When we are measuring position, we find a particle, and when we are measuring momentum, we find a wave. We cannot measure for both properties at the same time.

Now, says Wheeler, because we can't use both pieces of detection equipment together, we have to decide whether we want to measure the photon as a wave or as a particle. But according to quantum mechanics, says Wheeler, we can decide whether the photon is a particle (goes through one slit) or a wave (probabilistically travels through both slits) *after* the photon has already gone through the plate having the slits! Particles are correlated in space *and* time. Not only do our measurements reach across space faster than light and "determine" the state of particles that we have not yet measured, but they can reach backwards in time to "justify" the present results! As Wheeler has said, " . . . a choice in the present can alter in an irretrievable way what we are entitled to say about the past."

Advanced Potentials

Our discussion of superluminal phenomena signalling backwards in time should actually come as no surprise; nothing in relativity theory, quantum mechanics, electrodynamics, or mechanics says that time must move in one direction. Still, we humans seem to perceive "time's arrow" in certain "obviously irreversible" processes, such as wave motion asymmetry: an electromagnetic wave (such as a radio wave) expands from the transmitting source into infinity, never to return. This typical electromagnetic phenomenon is mathematically described in a solution to the moving charge equations (derived from James Clerk Maxwell's well-known electromagnetic field equations) as $(t + r/v)$, where t is time, r is the distance of the field from a moving charge, and v is the phase velocity. This is known as the *retarded* potential solution. The resultant type of wave motion is called retarded wave motion, since the electromagnetic fluctuation can be detected at a point in space distant from the origin after a period of time.

But as is the case with homogeneous differential equations (like the one P.A.M. Dirac discovered that defined the existence of both electrons *and* their time-reversed counterparts, positrons), there is another solution to the field equation: $(t - r/v)$. This is known as the *advanced* potential solution, resulting in advanced wave motion. Whereas retarded wave motion is when a wave travels from the origin outward to infinity, advanced wave motion is when a wave travels from infinity and collapses into the origin. This implies that reverse causality is real and that the wave can be detected at a remote location before it has been generated at the source!

Strangely, although this runs counter to common sense, Maxwell's equations and the laws of propagation do not favor ordinary retarded potential solutions over the more bizarre, time-reversed, advanced potential solutions. Indeed, N. Anderson, in his book *The Electromagnetic Field* (New York: Plenum, 1968), writes that, "Advanced potentials are now receiving a great deal of attention since they seem to be a means of avoiding some of the difficulties which beset electromagnetic theory. Advanced potentials were first invoked to try to solve the problem of obtaining an equation of motion for an electron moving in an electromagnetic field, which would take into consideration the radiation reaction, which is the force which acts on an electron due to its own electromagnetic field. Attempts using retarded potentials only are unsuccessful...."

Normally, the advanced solution is discarded, simply because no one has ever observed various parts of the distant universe "conspiring" to transmit incoming waves to radio and TV transmitters here on earth. But why doesn't this phenomenon occur? After all, it is known that various parts of the universe could be correlated in such a manner, since quantum mechanics and the EPR paradox state that all particles in the universe *do* somehow possess superluminal connections.

The best explanation centers on the fact that the universe is expanding rather than contracting or remaining stationary, so the electromagnetic radiation emanated by distant points of the universe is so red-shifted that it never reaches us. The expanding universe would even explain why time appears to travel in one direction: sources of energy can only be transmitters, never absorbers. Thus the apparent time asymmetry of the second law of thermodynamics and the "irreversible processes" that we observe associated with it are also explained. Systems can never return to their initial energy states (let alone *higher* energy states) because the universe is expanding and, as entropy increases, energy must be "diluted" more and more. An open container of liquid will evaporate away, never to return, and waves are only detectable when radiating from their sources since they never reach the other end of the universe – it keeps on expanding!

Man-Machine Communication

This should all be rather exciting to those who might one day build a computer based upon oddities in quantum theory and/or electrodynamics, allowing it to transcend the physical world in terms of processing speed. The "practical" barrier of about 1.5 trillion operations per second will be broken, but such superluminal computers will cause users additional problems.

For example, with so many signals spending a portion of their existence traveling faster than light, the large number of "negative cycles" of the processor clock that are necessary for the solution of increasingly difficult problems will result in the final answers appearing on the CRT screen or printer farther and farther back in time. At some point answers will begin appearing before the user has had a chance to key the question in!

To illustrate, let's say I wanted to ask a natural language processing AI program the question, "What prime number is closest to a trillion?" If I sat down at a terminal and keyed in this question, the answer would appear on the screen just *before* I sat down. A little disconcerting, but tolerable. Note that even though I already have the answer I (or somebody else) must at some point key in the question; otherwise, the question is never given to the processor and I will not have the answer! Sounds paradoxical, doesn't it? But it is perfectly logical. Causation is still causation, whether it be backward or forward.

But let's say that the answer appeared, and I felt particularly lazy that day and decided that I was not under any circumstances going to key in that question! What would happen? Would the universe come to an end? Would Dr. Who materialize and give me a strong lecture on space-time misbehavior?

No. When one thinks of the process only in terms of cause and effect, momentarily forgetting the direction of time, one realizes that the appearance of an answer on the screen means that a question *is* keyed into the keyboard, regardless of "time's arrow." Yours Truly or someone else *must* key in the question in the future, because now the answer is on the screen here in the past. This does not affect the idea of free will; in the future we are perfectly free to choose whether or not we want to key in the question (for example, we could decide by flipping a coin). But if we don't ask a question, of course, we

won't receive an answer in the past.

The longer it takes to work on a problem, the farther back in time the computer will answer. A truly difficult problem would take so many cycles of "negative time" to process that the answer would appear long before the question had even been formulated in the user's mind!

Now let's say that I asked the computer whether Gödel's proof is valid under all possible higher-order logics. The solution to that problem would take a great deal of negative time for the processor to come up with an answer. Perhaps years ago I saw the answer on the screen but dismissed it as a system malfunction, since I had not yet formulated the question in my mind or even knew what Gödel's proof was. Perhaps the negative time required for computing the answer is longer than the time the computer has been operational, so one never gets to see the answer! From the superluminal processor's "reversed-time" point of view, the computer would appear to have been disassembled while in the process of determining the answer. To our forward-time view, however, the computer had simply been assembled after the answer would have appeared on the screen, which would itself not have been manufactured yet. The first person to plug in the new computer, therefore, would be shocked to find the machine already working on a problem that it could never solve, since the system was not operational far enough back in time.

These problems could be alleviated – and new problems created – by typing the question into a memory buffer, just like a batch system, where a conventional computer and AI program would analyze the parameters of the question and estimate the appropriate amount of time to wait before sending the question to the superluminal processor. If the estimate were accurate, the delay period of submitting the question to the superluminal processor would be just a little longer than the negative time required for the answer. So if one keyed in a question, the answer would appear immediately afterward, even though the question had just been given to the delay buffer and not yet to the superluminal processor.

Of course, the more difficult the question, the longer it has to be kept in the delay memory buffer before being forwarded to the superluminal processor. If the user does not get an immediate response, then it means that the computer *will* be experiencing some mechanical difficulty between the time the question is keyed in and the time it is finally given to the superluminal processor. A mechanical problem or power failure is more damaging to a superluminal computer than it is to a conventional computer; although a physical interruption is the same whether one is looking at it while moving forward or backward in time, the superluminal computer user must wait for the failure to occur in order to fix it. Preventive maintenance would come in handy at this point! Normally, the user would have to wait until the mechanical problem actually occurs before keying in the question. The computer is thus not only a diagnostician of its own mechanical problems but a *precognosticator* of its future mechanical problems: the cessation of the computer's activity that prevents the answer's appearance represents another case of reverse causality, or information traveling backwards in time.

Brain-Processor Communication

With the sixth generation computer, the ultimate barrier between man and machine will finally be broken, eliminating the keyboard in favor of direct communication between the human brain and the computer. Experiments along this line have already been done, funded in some cases by CIA-type organizations in their attempts to develop a practical "thought-scanner."

When a human brain is exposed to some stimulus (such as a flash of light or the appearance of an alphanumeric symbol or an entire word), the segment of brain waves recorded by an electroencephalogram (EEG) about half a second later contains a waveform (the "evoked potential" or "event-related potential") that is a mental representation of the stimulus. The actual waveform is hidden, however, by "noise" – the unique spurious fluctuations introduced to the waveform by the individual brain undergoing analysis. The noise is eliminated and the actual waveform extracted by repeating the stimulus 50 or 100 times and then averaging all of the resulting recorded segments (each one slightly different from all the others because of the noise) and subjecting them to a Fourier analysis, which demonstrates the mathematical relationship between a complex periodic phenomenon and the simple harmonics that make it up.

The noise appears as harmonics of such complexity that for practical purposes it can be considered random noise; it cancels out once 100 samples are averaged. The final classification of the waveform includes not only the actual waveform with its fundamental wave and harmonics but the "latency" or length of time between the stimulus and the waveform's appearance.

Early success came in 1964, when W. Grey Walter and his colleagues at the Burden Neurological Institute in Bristol, England, discovered a wave that occurs whenever a subject anticipates something pleasurable (the so-called "expectancy wave"). In the same year, Samuel Sutton of the New York State Psychiatric Institute discovered the wave that we all produce when we hear a phone or doorbell or see a sudden flash of light – the so-called "surprise wave." Later, Helen Neville of the Salk Institute for Biological Studies in La Jolla, California, found the wave given off whenever one focuses one's attention on one stimulus out of many – the "selective attention wave."

In 1980, Steven Hillyard and Marta Kutas of the University of California, San Diego, discovered a negative polarity wave with a latency of 400 milliseconds (N400 wave) that represents a confused reaction to something – the "double-take wave." Thus, a person's learning process can be monitored simply by noting when during a programmed instruction course the person's brain emits the N400 wave representing his or her confusion.

In terms of identifying the language functions, neurophysiologist Donald York and speech pathologist Thomas Jensen at the University of Missouri have found "motor template waves" associated with about 20 different syllables, and a Russian scientist has supposedly isolated specific waves for specific meanings. His theory is that the different waveforms representing, for example, *chair*, *desk*, and *table* are special variations of an underlying waveform representing the concept of *furniture*, which itself could be a particular variation of a still more general waveform representing the concept of *object* or *thing*.

Practical applications of computer brain wave analysis have already appeared. Erich Sutter at the Smith-Kettlewell Institute in San Francisco has designed a system where the user dons some electrodes and glances at a particular flashing light when the user wants the computer to command a servomechanism. Each light, flashing in a pattern unique from all other lights, is delegated to specific tasks. The distinctive waveform resulting from the particular flashing-light stimulus is extracted from the brain wave pattern by a computer and is compared to the library of waveforms in the computer's memory through another pattern-matching technique. When a match occurs, the program "knows" which light was observed by the user, and a corresponding I/O subroutine is triggered to activate a certain piece of equipment. This system can help the handicapped operate bionic extremities or enable pilots to maneuver aircraft while subjected to paralyzing g-forces.

All of this work is remarkable when one realizes that less than 10 percent of the brain cells move electrical charges of sufficient strength to generate electromagnetic waves and that the average EEG records only about 32 channels. The electrical firings of the other neurons remain unanalyzed, although Robert Thatcher of the University of Maryland has described a scheme whereby a set of microwave transmitters would surround the head of an individual and the inter-

ference patterns produced between these beams and the brain's own waves would be recorded, processed by the computer, and played back as a 3-D moving picture of brain states and, therefore, mental processes – the perfect thought scanner.

Indeed, Dr. Glen Cartwright, a computer scientist and educational psychologist at McGill University, has already dubbed future brain-computer interactive devices "symbionic minds" (*sym*, as in *symbiosis*, and *bionic*).

By wearing a specially designed helmet, the sixth generation computer user will be able to locate and communicate with any person or piece of data simply by thinking about it. But why should the user be limited to a particular location, wearing a helmet that is plugged into a computer? Why not totally remove all the burdens of physical instrumentation from the user by having the computer read his brain waves remotely, without anything being worn by the user at all!

After all, the longer the wavelength of an electromagnetic wave, the farther it will travel before losing its attenuation. As it turns out, the brain produces waves of a remarkably low frequency. These are the delta waves (1 to 2 cycles per second), theta waves (3 to 6 cycles per second), alpha waves (7 to 14 cycles per second), and beta waves (15 to 30 cycles per second).

A 5 cycle-per-second electromagnetic wave (such as a brain wave) loses only 5 percent of its total energy after traveling 10,000 kilometers. If a human cranium radiates a millionth of a watt at 5 cycles per second, one would measure about 10^{-24} watts per square centimeter on the other side of the world. This doesn't sound like much until one realizes that radio astronomers have detected remnant background radio noise from the Big Bang, emanating from deep in space, which is only about 10^{-40} watts per square centimeter.

Additionally, experiments in global communications with submarines indicate that some of these frequencies set up resonances; the timing is such that a wave, weakened from its round-the-world journey, is reinforced upon its return by the next broadcast wave. The lowest of these frequencies are 7.8, 14.1, and 20.3 cycles per second, two of these falling within the alpha region and one in the beta region, the realm of ordinary waking consciousness!

Conversely, electromagnetic waves can be broadcast from the computer to the human brain to complete the communication. Physicist John Taylor has remarked that, "An aggregate of a thousand million nerve cells, each a millimeter in length, could act as a folded aerial with a total length of over a thousand kilometers." Muscle and nerve reactions to electromagnetic radiation between 50 and 200 cycles per second have reportedly been discovered in humans.

This kind of brain-computer communication would require huge antennas at the computer's location in order to receive the brain waves. We may, therefore, take one last step and apply the EPR effect of quantum mechanics to the problem. Instead of communicating by electromagnetic waves, the most advanced versions of the sixth generation computer (appearing sometime around the middle of the next century) will allow direct EPR coupling between the electrons in the human brain and the electrons in the computer's "transmitter/receiver."

In the 1951 edition of David Bohm's *Quantum Theory*, he comments on the idea of quantum aspects of brain processes that was put forth in an earlier work by Bohr (*Atomic Theory and the Description of Nature*): "In addition to such a classically describable mechanism that seems to act like a general system of communications [within the brain], Bohr's suggestion involves the idea that certain key points controlling this mechanism (which are, in turn, affected by the actions of this mechanism) are so sensitive and delicately balanced that they must be described in an essentially quantum-mechanical way. (We might, for example, imagine that such key points exist at certain types of nerve junctions.)"

One could now turn all of this around and ask if a sixth generation computer would, in fact, be conscious, since its "transmitter/receiver" would consist of halves of multiple two-particle systems correlated with their twin particles (and hence the mental processes) in the user's brain. This may be a meaningless question, since we do not even have a hint of what consciousness is in the brain.

Still, one is almost forced to speculate: are the laws of thought, even emotions and appreciations of beauty, reducible to physics, to special limiting cases of quantum theory? The pertinent facts could be distressing to those who would like to preserve a unique aspect of themselves despite the lessons of Copernicus, Darwin, and Freud, but fortunately for myself, I have just run out of editorial space.

■■J

Bibliography

Anderson, N. *The Electromagnetic Field*. New York: Plenum, 1968.

Ashby, R. "Some Consequences of Bremermann's Limit for Information Processing Systems," in *Cybernetic Problems in Bionics* (Bionics Symposium, 1966). Edited by H. L. Oestreicher and D. R. Moore, New York: Gordon & Breach, 1968.

Bell, J. S. "On the Problem of Hidden Variables in Quantum Mechanics," *Review of Modern Physics*, 38 (1966). An elaboration of some ideas contained in his original paper as found in: *Physics*, 1 (1964-5) 195.

Birkhoff, Garrett; Von Neumann, John. "The Logic of Quantum Mechanics," *Annals of Mathematics*, 37 (1936).

Bledsoe, W. W. "A Basic Limitation of the Speed of Digital Computers," *IRE Trans. Electr. Comp.*, EC-10 (1961) 530.

Bohm, David. *Quantum Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1951.

Born, Max. (Ed.) *The Born-Einstein Letters*. New York: Walker, 1971. See pp. 164-65, 168-76, 178, 188-89, 214-15.

Bremermann, H. J. "Part I: Limitations on Data Processing Arising from Quantum Theory," in "Optimization Through Evolution and Recombination," in *Self-Organizing Systems*. Edited by M. C. Yovits, G. T. Jacobi and G. D. Goldstein, Washington, D. C.: Spartan Books, 1962.

Bremermann, H. J. "Quantum Noise and Information," in *Proc. Fifth Berkeley Sympos. Math. Stat. a. Prob.*, Berkeley, CA: University of California Press, 1967.

Bremermann, H. J. "Complexity of Automata, Brains and Behavior," in *Physics and Mathematics of the Nervous System*. Edited by M. Conrad, W. Güttinger and M. Dal Cin; *Biomathematics Lecture Notes*, Vol. 4, Heidelberg: Springer Verlag, 1974.

Clauser, J. F.; Horne, M. A.; Shimony, A.; Holt, R. A. "Proposed Experiment to Test Hidden-Variable Theories," *Physics Review* (Letters) 23 (1969).

Clauser, John F.; Shimony, Abner. "Bell's Theorem: Experimental Tests and Implications," *Reports on Progress In Physics*, 41 (1978) 1881-1927.

Costa de Beauregard, O. *Revue Internationale de Philosophie*, 1: no. 61-62 (1962), and his subsequent paper found in *Dialectica*, 19 (1965) 280.

Davies, P. C. W. *The Physics of Time Asymmetry*. London: Surrey University Press, 1974.

Einstein, Albert. "Autobiographical Notes," in *Albert Einstein, Philosopher-Scientist*. Edited by Paul Schilpp, New York: Harper and Row, 1949.

Einstein, Albert; Podolsky, Boris; Rosen, Nathan. "Can Quantum-Mechanical Description of Reality Be Considered Complete?" *Physical Review*, 47 (1935) 777-780.

Freedman, Stuart J.; Clauser, John F. "Experimental Test of Local Hidden Variable Theories," *Physical Review Letters*, 28 (1972) 938ff.

Hall, Joseph; Kim, Christopher; McElroy, Brien; Shimony, Abner. "Wave-Packet Reduction as a Medium of Communication," *Foundations of Physics*, 7 (October 1977) 759-767.

Heisenberg, Werner. *Physics and Beyond*. New York: Harper and Row, 1971.

Hooker, Clifford A. "Concerning Einstein's, Podolsky's, and Rosen's Objection to Quantum Theory," *American Journal of Physics*, 38 (July 1970) 851-857.

Hooker, Clifford A. "The Nature of Quantum Mechanical Reality: Einstein Versus Bohr," In *Paradigms and Paradoxes*. Edited by Robert G. Colodny, Pittsburgh: University of Pittsburgh Press, 1972.

Knuth, D. E. "Mathematics and Computer Science: Coping with Finiteness," *Science*, 194 (1976) 1235-1242.

Kolata, Gina. "Does Gödel's Theorem Matter to Mathematics?" *Science*, 218 (1982) 779-780.

Popper, Karl. *The Logic of Scientific Discovery*. New York: Basic Books, 1959. See pp. 244-45, 444-48, 457-64.

Present, Gerald. "Understanding How the Universe Came Into Being: An Interview with John Archibald Wheeler," *Science Digest*, 86 (October 1975) 38-45.

Reichenbach, H. *The Direction of Time*. Berkeley: University of California Press, 1971.

Schrödinger, Erwin. "Discussions of Probability Relations between Separated Systems," *Proceedings of the Cambridge Philosophical Society*, 31 (1935) 555-562.

Selden, Gary. "Machines that Read Minds," *Science Digest*, 89 (October 1981) 60ff.

Stapp, Henry P. "The Copenhagen Interpretation and the Nature of Space-Time," *American Journal of Physics*, 40 (1972) 1098.

Stapp, Henry P. "Bell's Theorem and World Process," *Il Nuovo Cimento*, 29B (1975) 271.

Stapp, Henry P. "Are Superluminal Connections Necessary?" *Il Nuovo Cimento*, 40B (1977) 191-205.

Stratton, J. A. *Electromagnetic Theory*. New York: McGraw-Hill, 1941.

Taylor, John G. *The New Physics*. New York: Basic Books, 1972.

Taylor, John G. *Superminds*. New York: Viking, 1975.

Von Neumann, John. *The Mathematical Foundations of Quantum Mechanics*. (trans. Robert T. Beyer) Princeton: Princeton University Press, 1955.

Zukav, Gary. *The Dancing Wu Li Masters: An Overview of the New Physics*. New York: William Morrow, 1979.

A New Library for Small-C

by James Hendrix and Ernest Payne

The last installment on Small-C (*DDJ* Nos. 74 and 75) presented a meager function library that failed to stress compatibility with the Unix libraries. That shortcoming was quickly pointed out, and a number of people have gone on to develop their own "standard" libraries.

This article describes one such library, which was developed jointly by the authors. It was implemented under CP/M 2.2 and provides full support for the Small-C compiler and those programs that it compiles. Virtually all of the Unix functions that apply to foreign environments have been included. Naturally, standard files with I/O redirection and Unix-style command-line argument passing are supported. The MACRO-80 package was chosen for the project.

Other than for the arithmetic and logic library (a load module in this library), which remains essentially as Ron Cain presented it in 1980, only about 20 lines of assembly language code exist in this implementation. That makes even the low-level system functions much easier to understand and to adapt to other environments.

Some changes were made to the compiler itself with this implementation, so it has been redesignated **Version 2.1**. Except for the library, however, the differences from Version 2 are minor, so we only mention them in passing:

(1) To reduce command-line clutter, a filename in the command line (not a redirection specification) causes the compiler to output to a file having the same name but with an extension of MAC; the default input file extension is C. If more than one file is specified, they are compiled into a single program bearing the name of the first file. If no filename is given, input and output are done on the standard input and output files, as before, and redirection may be used to change the default console assignments.

(2) Undeclared functions are automatically declared to be external.

(3) The syntax (*func) ( ) for declaring pointers to functions and for calling such functions is now accepted.

(4) To accommodate the MACRO-80 package, the code-generating logic was changed and some functions were renamed to avoid clashes with reserved symbols and because MACRO-80 limits external names to six characters.

(5) Fixes and enhancements reported in *DDJ* by Andrew Macpherson and Paul West have been applied, along with a number of other minor fixes.

(6) Calls to nonstandard functions have been replaced by standard library calls.

A book entitled *The Small-C Handbook* is being printed by the Reston Publishing Company and should be available about the time you read this. It fully documents, from the user's point of view, the Small-C language and compiler, including this library. Most of the material in this article's section "User Functions" is borrowed from the book.

Copyright © 1984 L. E. Payne and J. E. Hendrix.

James Hendrix, Box 8378, University, MS 38677-8378.
Ernest Payne, 1331 W. Whispering Hills Dr., Tucson, AZ 85704.

Unix is a trademark of AT&T Bell Labs. MACRO-80 is a trademark of Microsoft Inc.

Library Organization

Generally, each library function is compiled and assembled separately; it is then kept in a library of relocatable object modules, which we call CLIB.REL. Some functions that share common code or are otherwise related are grouped into a single module. Examples are printf, fprintf, and the low-level system functions found in the module CSYSLIB.

At link time, L80 is directed to search CLIB.REL to resolve external references. Everything needed to support the program under CP/M is loaded and linked into the resulting COM file. Modules that are not referenced are not loaded. The minimum set of functions loaded comes to 5.5K bytes.

Since L80 does not scan backwards to find a module, the library is organized so that backward references only involve modules that are known to be loaded; otherwise, the library is arranged alphabetically. The compiler always generates an external reference to _link:

```
_link: : ext _main
         end
```

This occurs first in the library and forces the loading of CSYSLIB, which follows. The last module in the library is CALL, the arithmetic and logic library. It is loaded last in order to establish the location where free memory begins.

System Functions

The low-level system functions in the source file CSYSLIB.C are shown in Listing One (page 60). The names of these functions and the global variables that they use begin with the underscore character to avoid clashes with user-written function and variable names. MACRO-80 accepts these despite a statement to the contrary in the documentation. We found, however, that older versions of MACRO-80 would not accept such names as external references.

Program Initiation and Termination

The last part of CALL contains the following code:

```
_end: lhld  6              ;get bdos address
      sphl                 ;use for base of stack
      lxi   h,_end         ;get start of free memory
      shld  _memptr##      ;use for memory allocation
      jmp   _main##        ;parse command line,
                              ;execute program
      end   _end
```

The label _end designates the end of the program and the beginning of free memory. As indicated by the last line, this is also where program execution begins. (L80 plants a jump to this address at the beginning of the user program.) This logic is executed once, after which its memory space becomes available for use by the program. It first sets SP to the base of BDOS, thus overlaying the CCP; then sets _memptr to the beginning of free memory; and finally jumps to _main to prepare for execution of the program.

The function _parse is called by _main to perform command-line parsing and I/O redirection. It first copies the CP/M

command line into a dynamically allocated buffer and then scans it, calling _field to isolate arguments and _redirect to alter the assignments of stdin and stdout (appending is supported). If a redirection open fails, the program aborts after displaying the letter R for "redirection error."

Finally, argc and argv are pushed onto the stack, and main is called to start program execution. On return, exit is called with a zero argument signifying successful completion. Of course, the program could also call exit directly and pass whatever error code it wishes; the error code, if it is nonzero, is written as a byte to the console. Any open files are closed and a warm start is performed.

The BDOS Interface

A bare-bones BDOS interface is provided as a function called _bdos. It takes two arguments: first the function code to be placed in the C register, then the value to be placed in the DE register pair before calling address 5. On return, HL (the primary register for Small-C) receives the CP/M return code from the A register.

This simple interface is sufficient to support the library functions. A more complete BDOS interface was described by Terje Bolstad in *DDJ* No. 80 (June 1983). His ideas could be applied here to provide more flexibility if that is desirable.

Memory Management

Memory is allocated in unlinked, contiguous blocks beginning at the end of the program. Each call to _alloc allocates one block of zeroed or uninitialized memory, depending on the value of the second argument. The standard functions malloc and calloc call _alloc.

Memory may be deallocated by calling free or cfree, but care must be taken to deallocate memory in the reverse order from which it was allocated. Deallocating memory simply places a new value in _memptr; everything above that address is considered free.

A function called avail may be called to find out how much memory lies between _memptr and the stack pointer. If there is a program/stack overlap, avail either returns zero or aborts the program after displaying the letter M for "memory error," as requested. Malloc and calloc abort this way if sufficient memory is unavailable.

File Management

Low-level Unix functions identify files by means of small integer values called file descriptors. Whereas the Unix Standard I/O Library uses a pointer to a file control structure, our library uses the file descriptor approach throughout, even though the library includes functions from the Standard I/O Library. The impact of this difference is negligible if one restricts file references to the values returned by the function fopen and the symbols stdin, stdout, and stderr (defined in the header file STDIO.H as 0, 1, and 2, respectively).

The symbol MAXFILES in CLIB.DEF determines how many files may be opened simultaneously. Seven integer arrays are dimensioned according to that value. They are:

_status[MAXFILES] – This is a bit-encoded status word indicating whether a file is open on the corresponding file descriptor; zero implies a closed condition. Separate bits authorize reading and writing, and there are bits for end-of-file and error conditions.

_device[MAXFILES] – This contains a nonzero code designating one of the CP/M logic devices when a nondisk file is opened on the corresponding file descriptor.

_fcbptr[MAXFILES] – When a disk file is first opened on a file descriptor, a standard CP/M file control block (FCB) is dynamically allocated; its address is kept here. When the file is closed, the FCB is saved for reuse. Recall that the memory allocation scheme is too primitive to allow indiscriminate freeing of memory. Likewise, the program must not free memory that was allocated before a disk file is opened.

_bufptr[MAXFILES] – As with the FCB, a buffer is allocated when a disk file is opened; its address is kept here. Buffers also are saved when a file is closed.

_chrpos[MAXFILES] – This is an offset to the next byte to be obtained from the corresponding buffer. Note that the current sector of a file is maintained in the random record number field in the FCB itself, and only random CP/M reads and writes are performed.

_dirty[MAXFILES] – A nonzero value here indicates that the corresponding buffer contains new data that needs to be written to disk.

_nextc[MAXFILES] – A character that has been pushed back into a file by ungetc is kept here. The value EOF (defined in STDIO.H) cannot be pushed back, so it indicates an empty bucket.

No attempt is made to save space in these arrays, since Small-C generates significantly more code to access character values than to access integers. Since these integer arrays are small, more space would be tied up in code than would be saved by making some of them character arrays.

The function _mode is used extensively to verify both that a file descriptor has a legal value and that the indicated file is open. If the file descriptor is valid, the status of its file is returned; otherwise, zero is returned.

The function _open is called by both fopen and freopen. It tries to open a file on a designated file descriptor. It verifies that the first character of the mode argument is either **r** (read), **w** (write), or **a** (append). If the filename is CON:, LST:, PUN:, or RDR:, it simply assigns the indicated logical device to the file descriptor and returns. Otherwise, it allocates an FCB and a buffer for the file, as necessary. It then calls _newfcb to validate the filename, force it to upper case, and initialize the FCB. Finally, it calls _bdos to open the file.

In the case of read mode, the first sector (128-byte CP/M record) of the file is automatically read into the buffer. In the case of write mode, if the file already exists, it is deleted before a new one is created. In the case of append mode, if the file does not exist, a new one is created; if the file does exist, it is opened, positioned at the beginning of the last block, then read to end-of-file by repeatedly calling fgetc. If a control-Z signals end-of-file, _chrpos is adjusted to begin writing at that position. Note that, while this approach avoids reading the entire file, a character stream file is presumed and embedded control-Z characters may be missed.

If a + follows the mode character (e.g., r+), an update mode is implied and _status is set to allow both reading and writing. This new feature of Unix/C is documented by C. D. Perez in *A Guide to the C Library for UNIX Users*. Apparently, under Unix/C one must call fseek or rewind when switching between read and write operations. Our library, however, permits unrestricted switching between reads and writes; each operation begins with the byte following the last one transferred. Since Small-C does not yet support long integers, we do not support fseek. Instead, cseek provides a seek to CP/M record boundaries.

When a read detects end-of-file, the EOF bit in _status is set, thereby disabling further reads; writes, however, are permitted. The EOF bit is cleared by a successful seek or rewind operation. Opening a file in write or append mode automatically sets the EOF bit. One may extend a file either by opening in append mode or by opening in read-update mode, reading to end-of-file, and then writing.

Unix performs only binary file transfers, and the end of a file is maintained as a pointer in the directory structure. It is up to the device drivers to translate between the newline char-

acter and the carriage return, line feed sequence. This scheme, however, cannot be followed under CP/M. First, there is no place in the CP/M file directory to store an end-of-file pointer. Second, in order to maintain ASCII file compatibility with other CP/M software, control-Z must be used to signal end-of-file, and the newline character must be translated to a carriage return, line feed sequence on output and vice versa on input.

Therefore, it was necessary to choose a means of discriminating between byte stream (binary) and character stream (ASCII) operations. It would not have violated the intent of the C developers to specify different, nonUnix open modes for this purpose. But we preferred to retain the standard open modes and distinguish between the I/O functions instead. In our library, calls to read, fread, write, and fwrite give binary transfers; all other calls (e.g., fgetc, fgets, etc.) give ASCII transfers. This makes Small-C programs upwardly compatible with Unix without changing the open modes.

Note that binary reads detect end-of-file only at the end of the last sector in a file. This is necessarily inconsistent with Unix, which can tie the end of a file to the byte.

Diagrammed in Figure 1 (page 54) are the principal functions involved in I/O transfers. Lines connecting the function names illustrate the possible flow of control. All input/output requests pass through either _read or _write. These perform only binary data transfers, one byte at a time. The logic for character stream operations is in fgetc and fputc, which in turn are called by the other character stream functions.

The functions _conin and _conout perform console communication. CP/M direct console I/O is used for all console communication. On output, this gives full control of the console device to the program. It also allows the program to poll the console (by means of the function poll) for operator input while writing data to the console.

Had conventional console I/O been used, CP/M would also poll for control characters, making it a matter of chance who would get an input character. Using CP/M direct console I/O meant that the customary keyboard input services (echo, rubout, etc.) had to be built into _conin and fgets. But, as a look at these functions will show, the cost was modest.

As their names imply, _getsec and _putsec transfer a sector of data between a buffer and the disk. They are called when buffers become empty or full.

Getting a sector involves first flushing the buffer to disk if it contains new data. Next, the random record number (RRN) of the FCB is advanced by calling _advance. Finally, the data is transferred by calling _sector (which calls _bdos). The end-of-file status is set if the attempt fails.

Writing a sector differs from reading by more than the direction of the data flow. The order of calls to _sector and _advance are reversed since the RRN now describes the position in the file where the newly filled buffer should go. After transferring the data, _newbuf is called to pad the buffer with control-Z characters in anticipation of further _write calls.

In keeping with the Unix concept of treating directories as ordinary files, CSYSLIB makes disk directories look like ASCII files of filenames (one to a line) to programs that read them. A file specifier consisting of only the drive identifier (e.g., B:) is taken to mean the directory on the indicated drive. The value X: indicates the default drive. Such "filenames" may be used for redirecting the standard input file (e.g., <B:) and they are accepted by the functions fopen and freopen. This feature is a compile option controlled by the symbol DIR in CSYSLIB; it takes an additional 0.3K bytes of memory. That seems a small price to pay for the flexibility it gives. Combined with a respectable set of Small-C "software tools," this feature makes the dynamic creation of SUBMIT files for performing multi-file operations a routine affair.

User Functions*

Listing Two (to be printed next month) shows the source code for the user-level functions. Most of them are patterned after Unix counterparts. Some, however, are unique to this library; these are designated as Small-C functions. A number of symbols defined in STDIO.H are mentioned. They are:

```
#define stdin    0    /* fd for standard input file  */
#define stdout   1    /* fd for standard output file */
#define stderr   2    /* fd for standard error file  */
#define ERR     -2    /* error condition return value */
#define EOF     -1    /* end-of-file return value  */
#define NULL     0    /* value of a null character  */
```

In addition, the abbreviation fd refers to a file descriptor.

Input/Output Functions

- **fopen (name, mode) char *name, *mode;**

This function attempts to open the file indicated by the null-terminated character string at name. Mode points to a string indicating the use for which the file is to be opened. The values for mode are:

"r" – read
"w" – write
"a" – append

Read mode opens an existing file for input. Write mode either creates a new file or opens an old file and truncates it to permit writing to start at the beginning of the file. Append mode allows writing to begin at the end of an existing file or at the beginning of a new one.

In addition, there are modes that allow file updating (both reading and writing). They are:

"r+" – update read
"w+" – update write
"a+" – update append

These modes are the same as their nonupdate counterparts in terms of their effect at open time, but they also allow switching between read and write modes by interleaving calls to input and output functions.

Unless the program performs a seek or rewind operation, the next read or write operation begins at the point where the previous one finished. If the attempt to open a file is successful, fopen returns an fd value for the open file; otherwise, it returns NULL. The returned fd is then used in subsequent input/output function calls to identify the file. Only the standard files may be used without first calling fopen.

- **freopen (name, mode, fd) char *name, *mode; int fd;**

This function closes the previously opened file indicated by fd and opens a new one whose name is in the null-terminated character string at name. As for fopen, mode points to a character string indicating the open mode. It returns the original value of fd if the attempt is a success or NULL upon failure either to close the old file or to open the new one. Note, however, since the fd for the standard input file is zero, there is no way of distinguishing success from failure in that case.

- **fclose (fd) int fd;**

This function closes the specified file. If any new data is being held in the file's buffer, this data is first written to disk.

* *Most of the material in this section is reprinted with permission of the publisher and the author from: James E. Hendrix,* The Small-C Handbook, *Reston Publishing Co., © 1984. Reston Publishing Co., 11480 Sunset Hills Road, Reston, VA 22090.*

It returns NULL on success or a nonzero value on error.

- **fgetc (fd) int fd;** (alias getc)

This function returns the next character from the file indicated by fd. If no more characters remain in the file or an error condition occurs, it returns EOF. The end of the file is detected when the implementation standard end-of-file character occurs or the physical end of the file is reached.

- **ungetc (c, fd) char c; int fd;**

This function logically (not physically) pushes the character c back into the file indicated by fd. The next read from that file will retrieve that character first. Only one character at a time may be held in waiting. This function returns the character itself on success; it returns EOF if a previously pushed character is being held or if c has the value of EOF (you cannot push EOF into a file). Performing a seek or rewind operation on a file causes a pushed character to be forgotten.

- **getchar ( )**

This function is equivalent to fgetc (stdin).

- **fgets (str, sz, fd) char *str; int sz, fd;**

This function reads up to sz-1 characters into memory from the file indicated by fd, starting at the address indicated by str. Input is terminated after transferring a newline character, and a null character is appended after the newline or in the last position if newline is not found. Fgets returns str for success; otherwise, it returns NULL for end-of-file or an error.

- **fread (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;**

This function reads into memory from the file indicated by fd cnt items of data, sz bytes in length, starting at the address indicated by ptr. A count of the actual number of items read is returned to the caller (this might be less than cnt if the end of the file was encountered). This function performs a binary transfer; it does not convert carriage return, line feed sequences into newline characters, and it has no special regard for end-of-file bytes. It recognizes only the physical end of the file. You should call feof to determine when the data is exhausted and ferror to detect error conditions.

- **read (fd, ptr, cnt) int fd, cnt; char *ptr;**

This function reads cnt bytes of data into memory from the file indicated by fd, starting at the address indicated by ptr. A count of the actual number of bytes read is returned to the caller. This might be less than cnt if the end of the file was encountered. This function performs a binary transfer; it does not convert carriage return, line feed sequences into newline characters, and it has no special regard for end-of-file bytes. It recognizes only the physical end of the file. You should call feof to determine for sure when the data is exhausted and ferror to detect error conditions.

- **gets (str) char *str;**

This function reads characters into memory from stdin, starting at the address indicated by str. Input is terminated when a newline character is encountered, but the newline itself is not transferred; a null character terminates the input string. Gets returns str for success and NULL for end-of-file or an error. Since this function may transfer any amount of data, you must check the size of the input string to verify that it has not gone beyond its allotted space.

- **feof (fd) int fd;**

This function returns a nonzero value if the file designated by fd has reached its end. Otherwise, it returns NULL.

- **ferror (fd) int fd;**

This function returns a nonzero value if the file designated by fd has encountered an error condition since it was opened. Otherwise, it returns NULL.

- **clearerr (fd) int fd;**

This function clears the error status for the file indicated by fd.

- **fputc (c, fd) char c; int fd;** (alias putc)

This function writes the character c to the file indicated by fd. It returns the character itself on success; otherwise, it returns EOF. If c is a newline character, then a carriage return, line feed pair is written.

- **putchar (c) char c;**

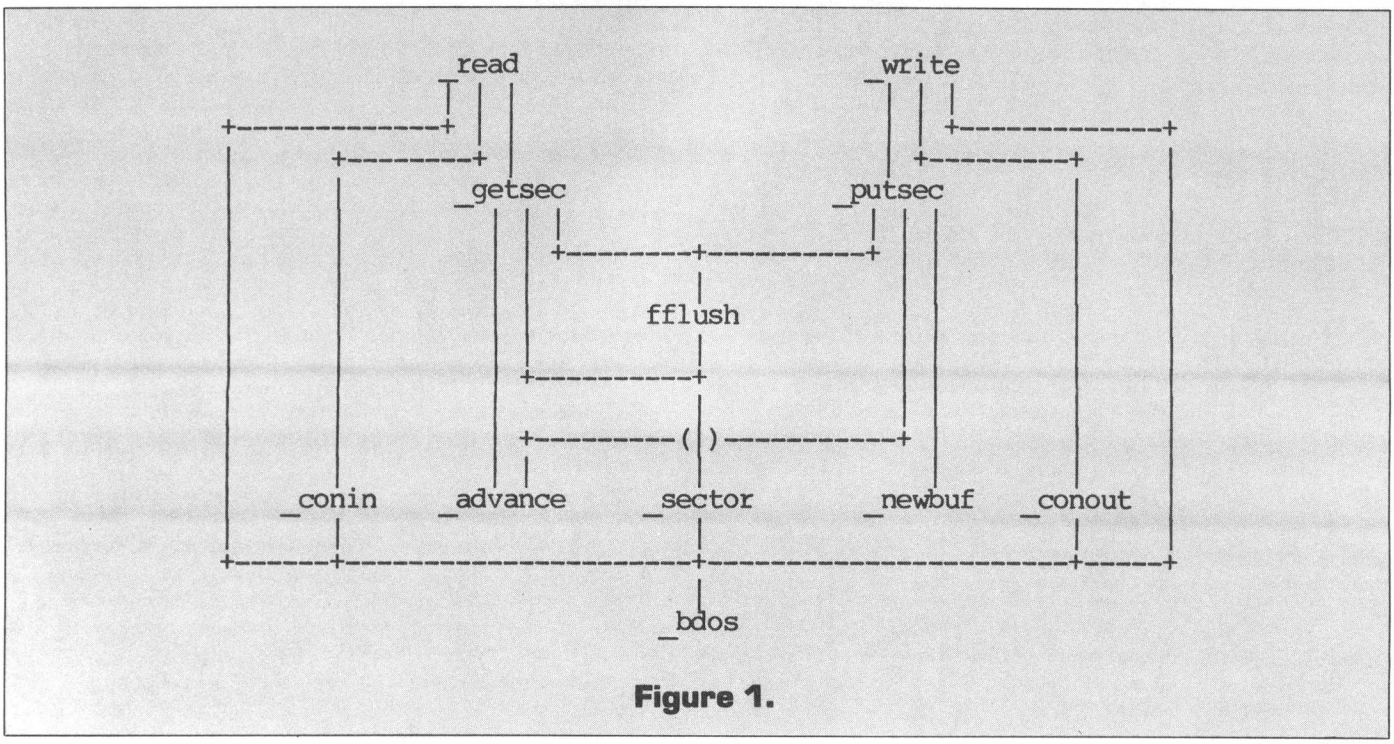


Figure 1.

This function is equivalent to fputc (c, stdout).

- **fputs (str, fd) char *str; int fd;**

This function writes characters beginning at the address indicated by str to the file indicated by fd. Successive characters are written until a null byte is found. The null byte is not written and a newline character is not appended.

- **puts (str) char *str;**

This function works like fputs (str, stdout) except that *it appends* a newline character to the output.

- **fwrite (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;**

This function writes from memory to the file indicated by fd cnt items of data, sz bytes long, starting at the address indicated by ptr. It returns a count of the number of items written. Although an error condition may cause the number of items written to be less than cnt, you should call ferror to verify all error conditions. This function performs a binary transfer; it does not convert newline characters into carriage return, line feed sequences.

- **write (fd, ptr, cnt) int fd, cnt; char *ptr;**

This function writes from memory to the file indicated by fd cnt bytes of data, starting at the address indicated by ptr. It returns a count of the number of bytes written. An error condition may cause the number of bytes written to be less than cnt. You should call ferror to verify error conditions, however. This function performs a binary transfer; it does not convert newline characters into carriage return, line feed sequences.

- **fflush (fd) int fd;**

This function forces any system-buffered changes out to the file. Ordinarily, data written to a disk file is held in a memory buffer until the buffer becomes full, the buffer space is needed to hold a different sector of data from the disk, or the file is closed. Fclose calls this function. Fflush returns NULL on success or EOF on error.

- **cseek (fd, offset, from) int fd, offset, from;**

This Small-C function positions the file indicated by fd to the beginning of the 128-byte record that is offset positions from the first record, current record, or end-of-file, depending on whether from is 0, 1, or 2, respectively. Subsequent reads and writes proceed from that point. It returns NULL for success and EOF otherwise.

- **rewind (fd) int fd;**

This function positions the file indicated by fd to its beginning. It is equivalent to a seek to the first byte of the file. It returns NULL on success and EOF otherwise.

- **ctell (fd) int fd;**

This Small-C function returns the position of the current record of the file indicated by fd. The returned value is the offset of the current 128-byte record with respect to the first record of the file. If fd is not assigned to a disk file, -1 is returned.

- **unlink (name) char *name;** (alias delete)

This function deletes the file indicated by the null-terminated character string at name. It returns NULL on success and ERR otherwise.

- **rename (old, new) char *old, *new;**

This Small-C function changes the name of the file specified by old to the name indicated by new. It returns NULL on success and ERR otherwise.

- **auxbuf (fd, size) int fd, size;**

This Small-C function allocates an auxiliary buffer of size bytes for fd. It returns zero on success and ERR on failure. Fd must be open, and size must be greater than zero and less than the amount of free memory. If fd is a device, the buffer is allocated but ignored. Extra buffering is useful in reducing disk head movement or drive switching during sequential operations. Once an auxiliary buffer is allocated, it sticks for the duration of program execution, even if fd is closed. Calling this function a second time for the same fd returns ERR but otherwise has no effect. Alternating read and write operations or performing seeks produces unpredictable results; ungetc( ), however, will operate normally. Ordinarily, it is counterproductive to allocate auxiliary buffers to both input and output files.

- **iscons (fd) int fd;**

This Small-C function returns a nonzero value if fd is assigned to the console; otherwise, it returns NULL.

- **isatty (fd) int fd;**

This function returns a nonzero value if fd is assigned to a device rather than a disk file; otherwise, it returns NULL.

Formatted Input/Output Functions

- **printf(str, arg1, arg2, . . .) char *str;**

This function writes to the standard output file a formatted character string consisting of the null-terminated character array at str, laced at specific points with the character-string equivalents of the arguments:

arg1, arg2 . . .

It also returns a count of the total number of characters written.

The string at str, which is called a control string, is required, but the other arguments are optional. The control string contains ordinary characters and groups of characters called conversion specifications. Each conversion specification informs printf how to convert the corresponding argument into a character string for output. The converted argument then replaces its conversion specification in the output. The character % signals the start of a conversion specification, and one of the letters b, c, d, o, s, u, or x ends it.

Between these may be found, in the order listed and with no intervening blanks, a minus sign, a decimal integer constant, and/or a decimal fraction. These subfields are all optional; in fact, one frequently sees conversion specifications with none of them. The minus sign indicates that the string, produced by applying a specified conversion to its argument, is to be left-adjusted in its field in the output. The decimal integer indicates the minimum width of that field (in characters); if more space is needed, it will be used, but at least the indicated number of positions will be generated. The decimal fraction is used where the argument being converted is itself a character string (or more correctly, the address of a character string). In this case the decimal fraction indicates the maximum number of characters to take from the string. If the specification has no decimal fraction, then all of the string is used.

The terminating letter indicates the type of conversion to be applied to the argument. It may be one of the following:

b The argument should be considered an unsigned integer and converted to *binary* format for output. No leading zeroes are generated. This specification is unique to Small-C and should be used with that in mind.

c The argument should be output as a *character* without any conversion, in which case the high-order byte will be ignored.

d The argument should be considered a signed integer and converted to a (possibly signed) *decimal* digit string for

output. No leading zeroes are generated. The sign is the leftmost character; it is blank for positive and "-" for negative.

o The argument should be considered an unsigned integer and converted to *octal* format for output. No leading zeroes are generated.

s The argument is the address of a null-terminated character *string* that should be output as is, subject to the justification, minimum width, and maximum size specifications indicated.

u The argument should be considered an unsigned integer and converted to an *unsigned* decimal character string for output. No leading zeroes are generated.

x The argument should be considered an unsigned integer and converted to *hexadecimal* format for output. No leading zeroes are generated.

If a % is followed by anything other than a valid specification, the % is ignored and the next character is written without change. So %% writes %.

Printf scans the control string from left to right, sending everything to stdout until it finds a % character. It then evaluates the conversion specification that follows and applies it to the first argument (following the control string). The resultant string is written to stdout. Printf then resumes writing data from the control string until it finds another conversion specification; it applies that one to the second argument. The procedure continues until the control string is exhausted. The result is a formatted output message consisting of both literal and variable data.

- **fprintf(fd, str, arg1, arg2, . . .) int fd; char *str;**

This function works like printf except that output goes to the file indicated by fd.

- **scanf(str, arg1, arg2, . . .) char *str;**

This function reads a series of fields from the standard input file, converts the fields to internal format according to conversion specifications contained in the control string str, and stores them at the locations indicated by the arguments:

arg1, arg2, . . .

It returns a count of the number of fields read.

A field in the input stream is a contiguous string of graphic characters. It ends with the next white-space character (blank, tab, or newline) unless its conversion specification indicates a maximum field width, in which case it ends when the field width is exhausted. A field normally begins with the first graphic character after the previous field; that is, leading white space is skipped.

Since the newline character is skipped while searching for the next field, scanf reads as many input lines as required to satisfy the number of conversion specifications in its control string. Each of the arguments following the control string must yield an address value.

The control string contains both conversion specifications and white space (which is ignored). Each conversion specification informs scanf how to convert the corresponding field into internal format, and each argument following str indicates the address where the corresponding converted field is to be stored. The character % signals the start of a conversion specification, and one of the letters b, c, d, o, s, u, or x ends it.

Between these may be found, with no intervening blanks, an asterisk and/or a decimal integer constant. As in printf, these subfields are both optional. The asterisk indicates that the corresponding field in the input stream is to be skipped; skip specifications do not have corresponding arguments. The numeric field indicates the maximum field width (in characters). If present, it causes the field to be terminated when the indicated number of characters has been scanned, even if no white space is found; however, if a white-space character is found before the field width is exhausted, the field is terminated at that point.

The terminating letter indicates the type of conversion to be applied to the field. It may be one of the following:

b The field should be considered a *binary* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small-C and should be used with that in mind.

c The field should be accepted as a single *character* without any conversion. This specification inhibits the normal skip over white-space characters. The argument for such a field should be a character address.

d The input field should be considered a (possibly signed) *decimal* integer and converted into an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored.

o The field should be considered an *octal* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored.

s The field should be considered a character *string* and stored with a null terminator at the character address indicated by its argument. There must be enough space at that address to hold the string and its terminator. (Remember, you can specify a maximum field width to prevent overflow.) The specification %1s will read the next graphic character, whereas %c will read the next character, whatever it is.

u The field should be considered an *unsigned* decimal integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small-C and should be used with that in mind.

x The field should be considered a *hexadecimal* number and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes or a leading 0x or 0X will be ignored.

Scanf scans the control string from left to right, processing input fields until the control string is exhausted or a field is found that does not match its conversion specification. If the value returned by scanf is less than the number of conversion specifications, an error has occurred or the end of the input file has been reached. EOF is returned if no fields are processed because end-of-file has been reached.

- **fscanf(fd, str, arg1, arg2, . . .) int fd; char *str;**

This function works like scanf except that the input is taken from the file indicated by fd.

Format Conversion Functions

- **atoi(str) char *str;**

This function converts the decimal number represented by the string at str to an integer and returns its value. Leading white space is skipped, and an optional sign (+ or -) may precede the leftmost digit. The first nonnumeric character terminates the conversion.

- **atoib(str, base) char *str; int base;**

This Small-C function converts the unsigned integer of base base, represented by the string at str, to an integer and returns its value. Leading white space is skipped. The first nonnumeric character terminates the conversion.

- **itoa(nbr, str) int nbr; char *str;**

This function converts the number nbr to its decimal character-string representation at str. The result is left-justified at str with a leading minus sign if nbr is negative. A null character terminates the string, which must be large

enough to hold the result.

- **itoab(nbr, str, base) int nbr; char *str; int base;**

This Small-C function converts the unsigned integer nbr to its character-string representation at str in base base. The result is left-justified at str. A null character terminates the string, which must be large enough to hold the result.

- **dtoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the (possibly) signed decimal number in the character string at str to an integer at nbr and returns the length of the numeric field found. The conversion stops when dtoi finds the end of the string or any illegal numeric character. When working with 16-bit integers, a leading sign and five digits at most will be used.

- **otoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the octal number in the character string at str to an integer at nbr and returns the length of the octal field found. It stops when it encounters a nonoctal digit in str. When working with 16-bit integers, six digits at most will be used.

- **utoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the unsigned decimal number represented by the character string at str to an integer at nbr and returns the length of the numeric field found. It stops when it encounters the end of the string or any nondecimal character. When working with 16-bit integers, five digits at most will be used.

- **xtoi(str, nbr) char *str; int *nbr;**

This Small-C function converts the hexadecimal number in the character string at str to an integer at nbr and returns the length of the hexadecimal field found. It stops when it encounters a nonhexadecimal digit in str. When working with 16-bit integers, four digits at most will be used.

- **itod(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to a signed (if negative) character string at str. The result is right-justified and blank-filled in str. The sign and possibly the high-order digits are truncated if the destination string is too small. It returns str. Sz indicates the length of the string. If sz is greater than zero, a null byte is placed at str[sz-1]. If sz is zero, a search for the first null byte following str locates the end of the string. If sz is less than zero, all sz characters of str are used, including the last one.

- **itoo(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to an octal character string at str. The result is right-justified and blank-filled in the destination string. High-order digits are truncated if the destination string is too small. It returns str. Sz functions the same as in itod.

- **itou(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to an unsigned decimal character string at str. It works like itod except that the high-order bit of nbr is taken for a magnitude bit.

- **itox(nbr, str, sz) int nbr, sz; char *str;**

This Small-C function converts nbr to a hexadecimal character string at str. In all other respects, it is identical to itoo.

String-Handling Functions

- **left(str) char *str;**

This Small-C function left-adjusts the character string at str. Starting with the first nonblank character and proceeding through the null terminator, it moves the string to the address indicated by str.

- **pad (str, ch, n) char *str, ch; int n;**

This Small-C function fills the string at str with n occurrences of the character ch.

- **reverse(str) char *str;**

This function reverses the order of the characters in the null-terminated string at str.

- **strcat(dest, sour) char *dest, *sour;**

This function appends the string at sour to the end of the string at dest. The null character at the end of dest is replaced by the leading character of sour. A null character terminates the new dest string. The space reserved for dest must be large enough to hold the result. This function returns dest.

- **strncat(dest, sour, n) char *dest, *sour; int n;**

This function works like strcat except that a maximum of n characters from the source string are transferred to the destination string.

- **strcmp(str1, str2) char *str1, *str2;**

This function returns an integer less than, equal to, or greater than zero, depending on whether the string at str1 is less than, equal to, or greater than the string at str2. Character-by-character comparisons are made, starting at the left end of the strings, until a difference is found. Comparison is based on the numeric values of the characters: str2 is considered less than str1 if str2 is equal to but shorter than str1, and vice versa.

- **lexcmp(str1, str2) char *str1, *str2;**

This Small-C function works like strcmp except that a lexicographic comparison is used. For meaningful results, only characters in the ASCII character set (0–127 decimal) should appear in the strings. Alphabetics are compared in dictionary order, with upper-case letters matching their lower-case equivalents. Special characters precede the alphabetics and are themselves preceded by the control characters except DEL, which compares highest.

- **strncmp(str1, str2, n) char *str1, *str2; int n;**

This function works like strcmp except that a maximum of n characters are compared.

- **strcpy(dest, sour) char *dest, *sour;**

This function copies the string at sour to dest; dest is returned. The space at dest must be large enough to hold the string at sour.

- **strncpy(dest, sour, n) char *dest, *sour; int n;**

This function works like strcpy except that n characters are placed in the destination string regardless of the length of the source string. If the source string is too short, null padding occurs; if it is too long, it is truncated in dest. A null character follows the last character placed in the destination string.

- **strlen(str) char *str;**

This function returns a count of the number of characters in the string at str. It does not count the null character that terminates the string.

- **strchr(str, c) char *str, c;**

This function returns a pointer to the first occurrence of the character c in the string at str. It returns NULL if the character is not found. Searching ends with the first null character.

- **strrchr(str, c) char *str, c;**

This function works like strchr except that the rightmost occurrence of the character is sought.

Character Classification Functions

The following functions determine whether or not a character belongs to a designated class of characters. They return *true* (nonzero) if it does and *false* (zero) if it does not.

- **isalnum(c) char c;**

  This function determines if c is alphanumeric (A-Z, a-z, or 0-9).

- **isalpha(c) char c;**

  This function determines if c is alphabetic (A-Z or a-z).

- **isascii(c) char c;**

  This function determines if c is an ASCII character (decimal values 0-127).

- **iscntrl(c) char c;**

  This function determines if c is a control character (ASCII codes 0-31 or 127).

- **isdigit(c) char c;**

  This function determines if c is a digit (0-9).

- **isgraph(c) char c;**

  This function determines if c is a graphic symbol (ASCII codes 33-126).

- **islower(c) char c;**

  This function determines if c is a lower-case letter (ASCII codes 97-122).

- **isprint(c) char c;**

  This function determines if c is a printable character (ASCII codes 32-126). Spaces are considered printable.

- **ispunct(c) char c;**

  This function determines if c is a punctuation character (all ASCII codes except control characters and alphanumeric characters).

- **isspace(c) char c;**

  This function determines if c is a white-space character (ASCII SP, HT, VT, CR, LF, or FF).

- **isupper(c) char c;**

  This function determines if c is an upper-case letter (ASCII codes 65-90).

- **isxdigit(c) char c;**

  This function determines if c is a hexadecimal digit (0-9, A-F, or a-f).

- **lexorder(c1, c2) char c1, c2;**

  This Small-C function returns an integer less than, equal to, or greater than zero depending on whether c1 is lexicographically less than, equal to, or greater than c2. For meaningful results, only characters in the ASCII character set (0-127 decimal) should be passed. Alphabetics are compared in dictionary order, with upper-case letters matching their lower-case equivalents. Special characters precede the alphabetics and are themselves preceded by the control characters except DEL, which compares highest.

Character Translation Functions

- **toascii(c) char c;**

  This function returns the ASCII equivalent of c. In systems that use the ASCII character set, it merely returns c unchanged. This function makes it possible to use the properties of the ASCII code set without introducing implementation dependencies into programs.

- **tolower(c) char c;**

  This function returns the lower-case equivalent of c if c is an upper-case letter; otherwise, it returns c unchanged.

- **toupper(c) char c;**

  This function returns the upper-case equivalent of c if c is a lower-case letter; otherwise, it returns c unchanged.

Mathematical Functions

- **abs(nbr) int nbr;**

  This function returns the absolute value of nbr.

- **sign(nbr) int nbr;**

  This function returns -1, 0, or +1, depending on whether nbr is less than, equal to, or greater than zero.

Program-Control Functions

- **calloc(nbr, sz) int nbr, sz;**

  This function allocates nbr*sz bytes of zeroed memory. It returns the address of the memory block on success and zero otherwise.

- **malloc(nbr) int nbr;**

  This function allocates nbr bytes of uninitialized memory. It returns the address of the memory block on success and zero otherwise.

- **avail(abort) int abort;**

  This Small-C function returns the number of bytes of free memory that are available between the program and the stack. It also checks to see whether the stack overlaps allocated memory; if so and if abort is not zero, the program is aborted, and the letter "S" is displayed on the console to indicate that a stack error has occurred. If abort is zero, however, avail returns zero to the caller. This function makes it possible to make full use of all available memory, but care should be taken to leave enough space for the stack to use.

- **free(addr) char *addr;** (alias cfree)

  This function frees up a block of allocated memory beginning at addr. It returns addr on success and NULL otherwise. It is necessary to free memory in the reverse order from which it was allocated. Freeing memory that was allocated before opening a file should be avoided since the open function dynamically allocates buffer and FCB space. You should not assume that closing a file relinquishes its space.

- **getarg(nbr, str, sz, argc, argv)**
  char *str; int nbr, sz, argc, *argv;

  This Small-C function locates the command-line argument indicated by nbr, moves it (null-terminated) to the string str of maximum size sz, and returns the length of the field obtained. Argc and argv must be the same values that are provided to the function main when the program is started. If nbr is zero, the program name is requested; if it is one, the first argument following the program name is requested; and so on. Because CP/M does not deliver the program name to a program, an asterisk is substituted in its place. If no argument corresponds to nbr, getarg puts a null byte at str and returns EOF.

- **poll(pause) int pause;**

  This Small-C function polls the console for operator input. If no input is pending, zero is returned. If a character is waiting, the value of pause determines what happens. If pause is zero, the character is returned immediately. If pause is not zero and the character is a control-S, there is a pause in program execution; when the next character is entered from the keyboard, zero is returned to the caller. If the character is a control-C, program execution is terminated. All other characters are returned to the caller immediately.

- **exit(errcode) int errcode;** (alias abort)

This function closes all open files and returns to the operating system. If errcode is not zero, it is written to the console: a program that exits with a control-G (bell), for instance, would sound the console beeper.

Availability

As with Version 2, copies of this implementation of Small-C are available for $25 (add $3 for overseas postage) in 8-inch, SSSD and NorthStar CP/M formats. There is no restriction on noncommercial distribution. A package of software tools written in Small-C is also available for $35. *The Small-C Handbook* may be obtained for $14.95. Inquiries about any of these items should be addressed to James Hendrix. For people interested in other formats and adaptations, James Hendrix is willing to act as a clearing house as long as the burden does not become too great. Please include a self-addressed, stamped envelope with your inquiry.

Conclusion

No doubt the Small-C compiler and its library will continue to develop as more and more people with access to the source code take an interest in it and report their developments. Some obvious areas for improvement in the library are:

(1) A better memory allocation scheme that permits allocation and deallocation operations to be performed in any order.

(2) Adapting this implementation to other CPUs and operating systems. This library should be especially easy to port to other environments because it is written in C and the logic seems easy to follow.

(3) Improvements in efficiency. No doubt this code can be made smaller and faster in many ways. We would hope, however, that such efforts would not obscure the simplicity and transparency of the logic.

We would like to express our appreciation to Ron Cain, who started the ball rolling in 1980, and to *Dr. Dobb's* for its continued support of Small-C.

■■J

Small-C Library (Text begins on page 50)

Listing One

```
 1 /*
 2 ** STDIO.H -- Standard Small-C Definitions
 3 **
 4 ** Copyright 1984  L. E. Payne and J. E. Hendrix
 5 */
 6 #define stdin    0
 7 #define stdout   1
 8 #define stderr   2
 9 #define ERR   (-2)
10 #define EOF   (-1)
11 #define YES      1
12 #define NO       0
13 #define NULL     0
14 #define CR      13
15 #define LF      10
16 #define BELL     7
17 #define SPACE
18 #define NEWLINE LF      /*23*/ /*45*/

 1 /*
 2 ** CLIB.DEF -- Definitions for Small-C library functions.
 3 **
 4 ** Copyright 1983  L. E. Payne and J. E. Hendrix
 5 **
 6 ** Credits:
 7 ** 1) This library of Small-C functions was produced
 8 **    jointly by:
 9 **
10 **    Ernest Payne
11 **    1331 W. Whispering Hills Drive
12 **    Tucson, AZ  85704
13 **
14 **    and
15 **
16 **    James E. Hendrix
17 **    Box 8378
18 **    University, MS  38677-8378
19 **
20 ** 2) The function _bdos() is an adaption of
21 **    Gene Cotton's work reported by Ron Cain (DDJ #48).
22 **
23 ** 3) The functions _parse(), _field(), and _redirect()
24 **    are a revision of Jan-Henrik Johansson's setarg()
25 **    (DDJ #74), and getarg() is a modification of his
26 **    revision of James Hendrix' function (DDJ #75).
27 **
28 ** 4) The standard C functions were obtained from
29 **    "A Guide to the C Library for UNIX User's
30 **    by C. D. Perez of Bell Laboratories.
31 **
32 */
33
34 /*
35 ** Definition of CP/M FCB and additional parameters
36 */
37 #define FCBSIZE   36  /* size of file control block */
38 #define DRIVE      0  /* CP/M drive designator offset */
39 #define NAMEOFF    1  /* CP/M file name offset */
40 #define NAMEOFF2  16  /* CP/M 2nd file name offset */
41 #define NAMESIZE   8  /* CP/M file name size */
42 #define TYPEOFF    9  /* CP/M file type offset */
43 #define TYPESIZE   3  /* CP/M file type size */
44 #define NTSIZE    11  /* CP/M file name & type size */
45 #define RRNOFF    33  /* CP/M random record number offset */
46 #define CPMEOF    26  /* CP/M end-of-file byte */
47 #define BUFSIZE  128  /* size of I/O buffer */
48 #define MAXFILES  10  /* maximum open files */
49 /*
50 ** CP/M function calls
51 */
52 #define CLOFIL   16  /* close file */
53 #define DCONIO    6  /* direct console i/o */
54 #define DELFIL   19  /* delete file */
55 #define FNDFIL   17  /* find first occurrence of a file */
56 #define FNDNXT   18  /* find next occurrence of a file */
57 #define GETPOS   36  /* get number of current sector */
58 #define GOCPM    00  /* go to CP/M */
59 #define LSTOUT   05  /* list output */
```

```
#define MAKFIL   22  /* make file */
#define OPNFIL   15  /* open file */
#define POSEND   35  /* position file to end */
#define PUNOUT   04  /* punch output */
#define RENAME   23  /* rename file */
#define RDRND    33  /* read sector randomly */
#define RDRINP   03  /* reader input */
#define SETDMA   26  /* set dma */
#define WRTRND   40  /* write sector randomly */
/*
** Device codes
*/
#define CPMCON DCONIO /* console */
#define CPMRDR RDRINP /* reader  */
#define CPMPUN PUNOUT /* punch   */
#define CPMLST LSTOUT /* list    */
/*
** File status bits
*/
#define RDBIT      1  /* open for read */
#define WRTBIT     2  /* open for write */
#define EOFBIT     4  /* eof condition */
#define ERRBIT     8  /* error condition */
/*
** ASCII characters
*/
#define ABORT    3
#define RUB      8
#define PAUSE   19
#define WIPE    24
#define DEL    127
```

```

/*
** CSYSLIB -- System-Level Library Functions
**
** Copyright 1984  L. E. Payne and J. E. Hendrix
*/

#include stdio.h
#include clib.def
#define NOCCARGC    /* no argument count passing */
#define DIR         /* compile directory option */

/*
****************** System Variables ********************
*/

int
 *_auxsz,            /* addr of _xsize[] in AUXBUF */
 *_auxef,            /* addr of _xeof[] in AUXBUF */
  _auxrd,            /* addr of _xread() in AUXBUF */
  _auxwt,            /* addr of _xwrite() in AUXBUF */
  _auxfl,            /* addr of _xflush() in AUXBUF */

  _cnt=1,            /* arg count for main */
  _vec[20],          /* arg vectors for main */

  _status[MAXFILES] = {RDBIT, WRTBIT, RDBIT|WRTBIT},
                     /* status of respective file */
  _device[MAXFILES] = {CPMCON, CPMCON, CPMCON},
                     /* non-disk device assignments */
  _nextc[MAXFILES]  = {EOF, EOF, EOF},
                     /* pigeonhole for ungetc bytes */
  _fcbptr[MAXFILES], /* FCB pointers for open files */
  _bufptr[MAXFILES], /* buffer pointers for files */
  _chrpos[MAXFILES], /* character position in buffer */
  _dirty[MAXFILES];  /* "true" if changed buffer */

char
 *_memptr,           /* pointer to free memory. */
  _arg1[]="*";       /* first arg for main */

/*
*************** System-Level Functions *****************
*/

/*
** -- Process Command Line, Execute main(), and Exit to CP/M
*/
_main() {
  _parse();
  main(_cnt,_vec);
  exit(0);
  }

/*
** Parse command line and setup argc and argv.
*/
_parse() {
  char *count, *ptr;
  count = 128;  /* CP/M command buffer address */
  ptr = _alloc(count = *count&255, YES);
  strncpy(ptr, 130, count-1);
  _vec[0]=_arg1;       /* first arg = "*" */
  while (*ptr) {
    if(isspace(*ptr)) {++ptr; continue;}
    switch(*ptr) {
      case '<': ptr = _redirect(ptr, "r", stdin);
                continue;
      case '>': if(*(ptr+1) == '>')
                     ptr = _redirect(ptr+1, "a", stdout);
                else ptr = _redirect(ptr,   "w", stdout);
                continue;
      default:  if(_cnt < 20) _vec[_cnt++] = ptr;
                ptr = _field(ptr);
      }
    }
  }

/*
** Isolate next command-line field.
*/
_field(ptr) char *ptr; {
  while(*ptr) {
    if(isspace(*ptr)) {
      *ptr = NULL;
      return (++ptr);
```

(Continued on page 64)

Small-C Library (Listing Continued, text begins on page 50)

Listing One

```
87        }
88      ++ptr;
89      }
90    return (ptr);
91    }
92
93  /*
94  ** Redirect stdin or stdout.
95  */
96  _redirect(ptr, mode, std)  char *ptr, *mode; int std; {
97    char *fn;
98    fn = ++ptr;
99    ptr = _field(ptr);
100   if(_open(fn, mode, std)==ERR) exit('R');
101   return (ptr);
102   }
103
104 /*
105 ** ------------ File Open
106 */
107
108 /*
109 ** Open file on specified fd.
110 */
111 _open(fn, mode, fd) char *fn, *mode; int fd; {
112   char *fcb;
113   if(!strchr("rwa", *mode)) return (ERR);
114   _nextc[fd] = EOF;
115   if(_auxef) _auxef[fd] = NO;
116   if(strcmp(fn,"CON:")==0) {
117     _device[fd]=CPMCON; _status[fd]=RDBIT|WRTBIT; return (fd);
118     }
119   if(strcmp(fn,"RDR:")==0) {
120     _device[fd]=CPMRDR; _status[fd]=RDBIT;  return (fd);
121     }
122   if(strcmp(fn,"PUN:")==0) {
123     _device[fd]=CPMPUN; _status[fd]=WRTBIT; return (fd);
124     }
125   if(strcmp(fn,"LST:")==0) {
126     _device[fd]=CPMLST; _status[fd]=WRTBIT; return (fd);
127     }
128   if(fcb = _fcbptr[fd]) pad(fcb, NULL, FCBSIZE);
129   else {
130     if((fcb = _fcbptr[fd] = _alloc(FCBSIZE, YES)) == NULL
131           || (_bufptr[fd] = _alloc(BUFSIZE, YES)) == NULL)
132         return (ERR);
133     }
134   pad(_bufptr[fd], CPMEOF, BUFSIZE);
135   _dirty[fd] = _device[fd] = _chrpos[fd] = 0;
136 #ifdef DIR
137   if(fn[1] == ':' && fn[2] == NULL) {  /* directory file */
138     pad(fcb, NULL, FCBSIZE);
139     pad(fcb+NAMEOFF, '?', NTSIZE);
140     if(toupper(fn[0]) != 'X') *fcb = toupper(fn[0]) - 64;
141     _chrpos[fd] = BUFSIZE;
142     _device[fd] = FNDFIL;
143     _status[fd] = RDBIT;
144     return (fd);
145     }
146 #endif
147   if(!_newfcb(fn,fcb)) return (ERR);
148   switch(*mode) {
149     case 'r': {
150       if(_bdos(OPNFIL,fcb)==255) return (ERR);
151       _status[fd] =  RDBIT;
152       if(_sector(fd,  RDRND)) _seteof(fd);
153       break;
154       }
155     case 'w': {
156       if(_bdos(FNDFIL,fcb)!=255) _bdos(DELFIL,fcb);
157     create:
158       if(_bdos(MAKFIL,fcb)==255) return (ERR);
159       _status[fd] = EOFBIT|WRTBIT;
160       break;
161       }
162     default: {      /* append mode */
163       if(_bdos(OPNFIL,fcb)==255) goto create;
164       _status[fd] = RDBIT;
165       cseek(fd, -1, 2);
166       while(fgetc(fd)!=EOF) ;
167       _status[fd] = EOFBIT|WRTBIT;
168       }
169     }
170   if(*(mode+1)=='+') _status[fd] |= RDBIT|WRTBIT;
171   return (fd);
172   }
173
174 /*
175 ** Create CP/M file control block from file name.
176 ** Entry: fn  = Legal CP/M file name (null terminated)
177 **              May be prefixed by letter of drive.
178 **       fcb = Pointer to memory space for CP/M fcb.
179 ** Returns the pointer to the fcb.
180 */
181 _newfcb(fn, fcb) char *fn, *fcb; {
182   char *fnptr;
183   pad(fcb+1, SPACE, NTSIZE);
184   if(*(fn + 1) == ':') {
185     *fcb = toupper(*fn) - 64;
186     fnptr = fn + 2;
187     }
188   else fnptr = fn;
189   if(*fnptr == NULL) return (NO);
190   fnptr = _loadfn(fcb + NAMEOFF, fnptr, NAMESIZE);
191   if(*fnptr == '.') ++fnptr;
192   else if(*fnptr) return (NO);
193   fnptr = _loadfn(fcb + TYPEOFF, fnptr, TYPESIZE);
194   if(*fnptr) return (NO);
195   return (YES);
196   }
197
198 /*
199 ** Load into fcb and validate file name.
200 */
201 _loadfn(dest, sour, max) char *dest, *sour; int max; {
202   while(*sour && !strchr("<>.,;:=?*[]", *sour)) {
203     if(max--) *dest++ = toupper(*sour++);
204     else break;
```

```
205     }
206   return (sour);
207   }
208
209 /*
210 ** ------------ File Input
211 */
212
213 /*
214 ** Binary-stream input of one byte from fd.
215 */
216 _read(fd) int fd; {
217   char *bufloc;
218   int ch;
219   switch (_mode(fd)) {
220     default: _seterr(fd); return (EOF);
221     case RDBIT:
222     case RDBIT|WRTBIT:
223     }
224   if((ch = _nextc[fd]) != EOF) {
225     _nextc[fd] = EOF;
226     return (ch);
227     }
228   switch(_device[fd]) {
229     /* PUN & LST can't occur since they are write mode */
230     case CPMCON: return (_conin());
231     case CPMRDR: return (_bdos(RDRINP,NULL));
232     default:
233         if(_auxsz && _auxsz[fd]) return (_auxrd(fd));
234         if(_chrpos[fd]>=BUFSIZE && !_getsec(fd))
235           return (EOF);
236         bufloc = _bufptr[fd] + _chrpos[fd]++;
237         return (*bufloc);
238     }
239   }
240
241 /*
242 ** Console character input.
243 */
244 _conin() {
245   int ch;
246   while(!(ch = _bdos(DCONIO, 255))) ;
247   switch(ch) {
248     case ABORT: exit(0);
249     case    LF:
250     case    CR: _conout(LF); return (_conout(CR));
251     case   DEL: ch = RUB;
252        default: if(ch < 32) { _conout('^'); _conout(ch+64);}
253                 else _conout(ch);
254                 return (ch);
255     }
256   }
257
258 /*
259 ** Read one sector from fd.
260 */
261 _getsec(fd) int fd; {
262 #ifdef DIR
263   if(_device[fd]) {        /* directory file */
264     char *bp, *name, *type, *end;
265     _bdos(SETDMA, 128);
266     if((name = _bdos(_device[fd], _fcbptr[fd])) == 255) {
267       _seteof(fd);
268       return (NO);
269       }
270     _device[fd] = FNDNXT;
271     name = (name << 5) + (128 + NAMEOFF);
272     type = name + NAMESIZE;
273     end = name + NTSIZE;
274     bp = _bufptr[fd] + BUFSIZE;
275     *--bp = CR;
276     while(--end >= name) { /* put filename at end of buffer */
277       if(*end == SPACE) continue;
278       *--bp = *end;
279       if(end == type) *--bp = '.';
280       }
281     _chrpos[fd] = bp - _bufptr[fd];
282     return (YES);
283     }
284 #endif
285   if(fflush(fd)) return (NO);
286   _advance(fd);
287   if(_sector(fd, RDRND)) {
288     pad(_bufptr[fd], CPMEOF, BUFSIZE);
289     _seteof(fd);
290     return (NO);
291     }
292   return (YES);
293   }
294
295 /*
296 ** ------------ File Output
297 */
298
299 /*
300 ** Binary-Stream output of one byte to fd.
301 */
302 _write(ch, fd) int ch, fd; {
303   char *bufloc;
304   switch (_mode(fd)) {
305     default: _seterr(fd); return (EOF);
306     case WRTBIT:
307     case WRTBIT|RDBIT:
308     case WRTBIT|EOFBIT:
309     case WRTBIT|EOFBIT|RDBIT:
310     }
311   switch(_device[fd]) {
312     /* RDR can't occur since it is read mode */
313     case CPMCON: return (_conout(ch));
314     case CPMPUN:
315     case CPMLST: _bdos(_device[fd], ch);
316                  break;
317     default:
318       if(_auxsz && _auxsz[fd]) return (_auxwt(ch, fd));
319       if(_chrpos[fd]>=BUFSIZE && !_putsec(fd)) return (EOF);
320       bufloc = _bufptr[fd] + _chrpos[fd]++;
321       *bufloc = ch;
322       _dirty[fd] = YES;
323     }
324   return (ch);
325   }
326
327 /*
328 ** Console character output.
329 */
330 _conout(ch) int ch; {
331   _bdos(DCONIO, ch);
332   return (ch);
```

(Continued on next page)

Small-C Library (Listing Continued, text begins on page 50)

Listing One

```
333   }
334
335 /*
336 ** Write one sector to fd.
337 */
338 _putsec(fd) int fd; {
339   if(fflush(fd)) return (NO);
340   _advance(fd);
341   pad(_bufptr[fd], CPMEOF, BUFSIZE);
342   return (YES);
343   }
344
345 /*
346 ** ------------ Buffer Service
347 */
348
349 /*
350 ** Advance to next sector.
351 */
352 _advance(fd) int fd; {
353   int *rrn;
354   rrn = _fcbptr[fd] + RRNOFF;
355   ++(*rrn);
356   _chrpos[fd] = 0;
357   }
358
359 /*
360 ** Sector I/O.
361 */
362 _sector(fd, func) int fd, func; {
363   int error;
364   _bdos(SETDMA, _bufptr[fd]);
365   error = _bdos(func, _fcbptr[fd]);
366   _bdos(SETDMA, 128);
367   _dirty[fd] = NO;
368   return (error);
369   }
370
371 /*
372 ** ------------ File Status
373 */
374
375 /*
376 ** Return fd's open mode, else NULL.
377 */
378 _mode(fd) char *fd; {
379   if(fd < MAXFILES) return (_status[fd]);
380   return (NULL);
381   }
382
383 /*
384 ** Set eof status for fd and
385 ** disable future i/o unless writing is allowed.
386 */
387 _seteof(fd) int fd; {
388   _status[fd] |= EOFBIT;
389   }
390
391 /*
392 ** Clear eof status for fd.
393 */
394 _clreof(fd) int fd; {
395   _status[fd] &= ~EOFBIT;
396   }
397
398 /*
399 ** Set error status for fd.
400 */
401 _seterr(fd) int fd; {
402   _status[fd] |= ERRBIT;
403   }
404
405 /*
406 ** ------------ Memory Allocation
407 */
408
409 /*
410 ** Allocate n bytes of (possibly zeroed) memory.
411 ** Entry: n = Size of the items in bytes.
412 **    clear = "true" if clearing is desired.
413 ** Returns the address of the allocated block of memory
414 ** or NULL if the requested amount of space is not available.
415 */
416 _alloc(n, clear) char *n; int clear; {
417   char *oldptr;
418   if(n < avail(YES)) {
419     if(clear) pad(_memptr, NULL, n);
420     oldptr = _memptr;
421     _memptr += n;
422     return (oldptr);
423     }
424   return (NULL);
425   }
426
427 /*
428 ** ------------ CP/M Interface
429 */
430
431 /*
432 ** Issue CP/M function and return result.
433 ** Entry: c  = CP/M function code (register C)
434 **        de = CP/M parameter (register DE or E)
435 ** Returns the CP/M return code (register A)
436 */
437 _bdos(c,de) int c,de; {
438 #asm
439         pop     h       ;hold return address
440         pop     d       ;load CP/M function parameter
441         pop     b       ;load CP/M function number
442         push    b       ;restore
443         push    d       ;  the
444         push    h       ;     stack
445         call    5       ;call bdos
446         mvi     h,0     ;
447         mov     l,a     ;return the CP/M response
448 #endasm
449   }
```

End Listing One

Listing Two

File: ABS.C

```
/*
** abs -- returns absolute value of nbr
*/
abs(nbr)  int nbr; {
  if(nbr < 0) return (-nbr);
  return (nbr);
  }
```

File: ATOI.C

```
#define NOCCARGC  /* no argument count passing */
/*
** atoi(s) - convert s to integer.
*/
atoi(s) char *s; {
  int sign, n;
  while(isspace(*s)) ++s;
  sign = 1;
  switch(*s) {
    case '-': sign = -1;
    case '+': ++s;
    }
  n = 0;
  while(isdigit(*s)) n = 10 * n + *s++ - '0';
  return (sign * n);
  }
```

File: ATOIB.C

```
#define NOCCARGC  /* no argument count passing */
/*
** atoib(s,b) - Convert s to "unsigned" integer in base b.
**              NOTE: This is a non-standard function.
*/
atoib(s, b) char *s; int b; {
  int n, digit;
  n = 0;
  while(isspace(*s)) ++s;
  while((digit = (127 & *s++)) >= '0') {
    if(digit >= 'a')      digit -= 87;
    else if(digit >= 'A') digit -= 55;
    else                  digit -= '0';
    if(digit >= b) break;
    n = b * n + digit;
    }
  return (n);
  }

```

File: AUXBUF.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
#include clib.def
extern int *_auxsz, *_auxef, _auxrd, _auxwt, _auxfl,
           _status[];
/*
** This module is loaded with a program only if auxbuf()
** is called.  It links to _open(), _read(), _write(), and
** fflush() through _auxsz, _auxef, _auxrd, _auxwt, and _auxfl
** in CSYSLIB.  This technique reduces the overhead for
** programs which don't use auxiliary buffering.  Presumably,
** if there is enough memory for extra buffering, there is
** room to spare for this overhead too.  A bug in some
** versions of Small-C between 2.0 and 2.1 may cause the calls
** to _auxrd, _auxwt, and _auxfl in _read(), _write(), and
** fflush(), respectively, to produce bad code.  The current
** compiler corrects the problem.
*/
int
  _xsize[MAXFILES],  /* size of buffer */
  _xaddr[MAXFILES],  /* aux buffer address */
  _xnext[MAXFILES],  /* address of next byte in buffer */
  _xend[MAXFILES],   /* address of end-of-data in buffer */
  _xeof[MAXFILES];   /* true if current buffer ends file */
/*
** auxbuf -- allocate an auxiliary input buffer for fd
**   fd = file descriptor of an open file
** size = size of buffer to be allocated
** Returns NULL on success, else ERR.
** Note: Ungetc() still works.
**       A 2nd call returns ERR, but has no effect.
**       If fd is a device, buffer is allocated but ignored.
**       Buffer stays allocated when fd is closed.
**       Do not mix reads and writes or perform seeks on fd.
*/
auxbuf(fd, size) int fd; char *size; {   /* fake unsigned */
  if(!_mode(fd) || !size || avail(NO) < size   || _xsize[fd])
    return (ERR);
  _xaddr[fd] = _xnext[fd] = _xend[fd] = malloc(size);
  _auxef = _xeof;     /* tell _open() where _xeof[] is */
  _auxrd = _xread;    /* tell _read() where _xread() is */
  _auxwt = _xwrite;   /* tell _write() where _xwrite() is */
  _auxsz = _xsize;    /* tell both where _xsize[] is */
  _auxfl = _xflush;   /* tell fflush() where _xflush() is */
  _xsize[fd] = size;  /* tell _read() that fd has aux buf */
  return (NULL);
  }

/*
** Fill buffer if necessary, and return next byte.
*/
_xread(fd) int fd; {
  char *ptr;
  while(YES) {
    ptr = _xnext[fd];
    if(ptr < _xend[fd]) {++_xnext[fd]; return (*ptr);}
    if(_xeof[fd]) {_seteof(fd); return (EOF);}
    _auxsz = NULL;          /* avoid recursive loop */
    _xend[fd] = _xaddr[fd]
              + read(fd, _xnext[fd]=_xaddr[fd], _xsize[fd]);
    _auxsz = _xsize;        /* restore _auxsz */
    if(feof(fd)) {_xeof[fd] = YES; _clreof(fd);}
    }
  }
```

(Continued on page 70)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

```
65
66 /*
67 ** Empty buffer if necessary, and store ch in buffer.
68 */
69 _xwrite(ch, fd) int ch, fd; {
70   char *ptr;
71   while(YES) {
72     ptr = _xnext[fd];
73     if(ptr < (_xaddr[fd] + _xsize[fd]))
74       {*ptr = ch; ++_xnext[fd]; return (ch);}
75     if(_xflush(fd)) return (EOF);
76     }
77   }
78
79 /*
80 ** Flush aux buffer to file.
81 */
82 _xflush(fd) int fd; {
83   int i, j;
84   i = _xnext[fd] - _xaddr[fd];
85   _auxsz = NULL;   /* avoid recursive loop */
86   j = write(fd, _xnext[fd]=_xaddr[fd], i);
87   _auxsz = _xsize; /* restore _auxsz */
88   if(i != j) return (EOF);
89   return (NULL);
90   }
```

File: AVAIL.C

```
1 #define NOCCARGC  /* no argument count passing */
2 extern char *_memptr;
3 /*
4 ** Return the number of bytes of available memory.
5 ** In case of a stack overflow condition, if 'abort'
6 ** is non-zero the program aborts with an 'S' clue,
7 ** otherwise zero is returned.
8 */
9 avail(abort) int abort; {
10   char x;
11   if(&x < _memptr) {
12     if(abort) exit('M');
13     return (0);
14     }
15   return (&x - _memptr);
16   }
17
```

File: CALLOC.C

```
1 #define NOCCARGC  /* no argument count passing */
2 #include stdio.h
3 /*
4 ** Cleared-memory allocation of n items of size bytes.
5 ** n     = Number of items to allocate space for.
6 ** size  = Size of the items in bytes.
7 ** Returns the address of the allocated block,
8 ** else NULL for failure.
9 */
0 calloc(n, size) char *n, *size; {
1   return (_alloc(n*size, YES));
2   }
```

File: CLEARERR.C

```
1 #define NOCCARGC  /* no arg count passing */
2 #include stdio.h
3 #include clib.def
4 extern int _status[];
5 /*
6 ** Clear error status for fd.
7 */
8 clearerr(fd) int fd; {
9   if(_mode(fd)) _status[fd] &= ~ERRBIT;
10  }
11
```

File: CSEEK.C

```
1 #define NOCCARGC  /* no argument count passing */
2 #include stdio.h
3 #include clib.def
4 extern int _fcbptr[], _chrpos[], _nextc[];
5 /*
6 ** Position fd to the 128-byte record indicated by
7 ** "offset" relative to the point indicated by "base."
8 **
9 **     BASE     OFFSET-RELATIVE-TO
10 **       0      first record
11 **       1      current record
12 **       2      end of file (last record + 1)
13 **
14 ** Returns NULL on success, else EOF.
15 */
16 cseek(fd, offset, base) int fd, offset, base; {
17   int oldrrn, *rrn;
18   if(!_mode(fd) || isatty(fd) || fflush(fd)) return (EOF);
19   rrn = _fcbptr[fd] + RRNOFF;
20   oldrrn = *rrn;
21   switch (base) {
22     case 2: _bdos(POSEND, _fcbptr[fd]);
23     case 1: *rrn += offset;
24             break;
25     case 0: *rrn = offset;
26             break;
27     default: return (EOF);
28     }
29   if(_sector(fd, RDRND)) {
30     *rrn = oldrrn;
31     return (EOF);
32     }
33   _chrpos[fd] = 0;
34   _nextc[fd] = EOF;
35   _clreof(fd);
36   return (NULL);
37   }
38
```

File: CTELL.C

```
1 #define NOCCARGC  /* no arg count passing */
2 #include stdio.h
```

```
3 #include clib.def
4 extern int _fcbptr[], _chrpos[];
5 /*
6 ** Return offset to current 128-byte record.
7 */
8 ctell(fd) int fd; {
9   int *rrn;
10  if(!_mode(fd) || isatty(fd)) return (-1);
11  rrn=_fcbptr[fd]+RRNOFF;
12  return (*rrn);
13  }
14 /*
15 ** Return offset to next character in current buffer.
16 */
17 ctellc(fd) int fd; {
18  return (_chrpos[fd]);
19  }
20
```

File: DTOI.C

```
1 #define NOCCARGC  /* no argument count passing */
2 #include stdio.h
3 /*
4 ** dtoi -- convert signed decimal string to integer nbr
5 **         returns field length, else ERR on error
6 */
7 dtoi(decstr, nbr)  char *decstr;  int *nbr;  {
8   int len, s;
9   if((*decstr)=='-') {s=1; ++decstr;} else s=0;
10  if((len=utoi(decstr, nbr))<0) return ERR;
11  if(*nbr<0) return ERR;
12  if(s) {*nbr = -*nbr; return ++len;} else return len;
13  }
```

File: EXIT.C

```
1 #define NOCCARGC  /* no argument count passing */
2 #include stdio.h
3 #include clib.def
4 /*
5 ** Close all open files and exit to CP/M.
6 ** Entry: errcode = Character to be sent to stderr.
7 ** Returns to CP/M rather than the caller.
8 */
9 exit(errcode) char errcode; {
10  int fd;
11  if(errcode) _conout(errcode);
12  for(fd=0; fd < MAXFILES; fclose(fd++));
13  _bdos(GOCPM,NULL);
14  }
15 #asm
16 abort  equ    exit
17        entry  abort
18 #endasm
```

File: FCLOSE.C

```
1 #define NOCCARGC  /* no argument count passing */
2 #include stdio.h
3 #include clib.def
4 /*
5 ** Close fd
6 ** Entry: fd = File descriptor for file to be closed.
7 ** Returns NULL for success, otherwise ERR
8 */
9 extern int _fcbptr[], _status[], _device[];
10 fclose(fd) int fd; {
11  if(!_mode(fd)) return (ERR);
12  if(!isatty(fd)) {
13    if(fflush(fd) || _bdos(CLOFIL,_fcbptr[fd])==255)
14      return (ERR);
15    }
16  return (_status[fd]=_device[fd]=NULL);
17  }
18
```

File: FEOF.C

```
1 #define NOCCARGC  /* no argument count passing */
2 #include clib.def
3 extern int _status[];
4 /*
5 ** Test for end-of-file status.
6 ** Entry: fd = file descriptor
7 ** Returns non-zero if fd is at eof, else zero.
8 */
9 feof(fd) int fd; {
10  return (_status[fd] & EOFBIT);
11  }
12
```

File: FERROR.C

```
1 #define NOCCARGC  /* no argument count passing */
2 #include stdio.h
3 #include clib.def
4 extern int _dirty[], *_auxsz, _auxfl;
5 /*
6 ** Write buffer for fd if it has changes.
7 */
8 ferror(fd) int fd; {
9   return (_status[fd] & ERRBIT);
10  }
```

File: FFLUSH.C

```
1 #define NOCCARGC  /* no arg count passing */
2 #include stdio.h
3 #include clib.def
4 extern _status[];
5 /*
6 ** Test for error status on fd.
7 ** Entry: fd = File descriptor of pertinent file.
8 ** Returns NULL on success, otherwise EOF.
9 */
10 fflush(fd) int fd; {
11  if(!_mode(fd)) return (ERR);
12  if(_auxsz && _auxsz[fd] && _auxfl(fd)) return (ERR);
13  if(!isatty(fd) && _dirty[fd] && _sector(fd, WRTRND)) {
14    _seterr(fd);
15    return (ERR);
16    }
17  return (NULL);
18  }
19
```

(Continued on next page)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

File: FGETC.C

```
#define NOCCARGC  /* no argument count passing */
#include stdio.h
#include clib.def
extern int _chrpos[];
/*
** Character-stream input of one character from fd.
** Entry: fd = File descriptor of pertinent file.
** Returns the next character on success, else EOF.
*/
fgetc(fd) int fd; {
  int ch;
  while(1) {
    switch(ch = _read(fd)) {
      default:     return (ch);
      case CPMEOF: switch(_chrpos[fd]) {
                     default: --_chrpos[fd];
                     case 0:
                     case BUFSIZE:
                     }
                   _seteof(fd);
                   return (EOF);
      case CR:     return ('\n');
      case LF:    /* NOTE: _conin() maps LF -> CR */
      }
    }
  }
#asm
getc equ   fgetc
     entry getc
#endasm

```

File: FGETS.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
/*
** Gets an entire string (including its newline
** terminator) or size-1 characters, whichever comes
** first. The input is terminated by a null character.
** Entry: str  = Pointer to destination buffer.
**        size = Size of the destination buffer.
**        fd   = File descriptor of pertinent file.
** Returns str on success, else NULL.
*/
fgets(str, size, fd) char *str; int size, fd; {
  return (_gets(str, size, fd, 1));
  }

/*
** Gets an entire string from stdin (excluding its newline
** terminator) or size-1 characters, whichever comes
** first. The input is terminated by a null character.
** The user buffer must be large enough to hold the data.
** Entry: str  = Pointer to destination buffer.
** Returns str on success, else NULL.
*/
gets(str) char *str; {
  return (_gets(str, 32767, stdin, 0));
  }

_gets(str, size, fd, nl) char *str; int size, fd, nl; {
  int backup;
  char *next;
  next = str;
  while(--size > 0) {
    switch (*next = fgetc(fd)) {
      case  EOF: *next = NULL;
                 if(next == str) return (NULL);
                 return (str);
      case '\n': *(next + nl) = NULL;
                 return (str);
      case  RUB: if(next > str) backup = 1; else backup = 0;
                 goto backout;
      case WIPE: backup = next - str;
        backout:
                 if(iscons(fd)) {
                   fputs("\b \b\b \b", stderr);
                   ++size;
                   while(backup--) {
                     fputs("\b \b", stderr);
                     if(*--next < 32) fputs("\b \b", stderr);
                     ++size;
                     }
                   continue;
                   }
        default: ++next;
      }
    }
  *next = NULL;
  return (str);
  }

```

File: FOPEN.C

```
#define NOCCARGC  /* no arg count passing */
#include stdio.h
#include clib.def
/*
** Open file indicated by fn.
** Entry: fn   = Null-terminated CP/M file name.
**               May be prefixed by letter of dirve.
**               May be just CON:, RDR:, PUN:, or LST:.
**        mode = "a"  - append
**               "r"  - read
**               "w"  - write
**               "a+" - append update
```

```
13 **              "r+" - read   update
14 **              "w+" - write  update
15 ** Returns a file descriptor on success, else NULL.
16 */
17 fopen(fn, mode) char *fn, *mode; {
18   int fd;
19   fd = 0; /* skip stdin (= error return) */
20   while(++fd < MAXFILES) {
21     if(_mode(fd) == NULL) {
22       if(_open(fn, mode, fd)!=ERR) return (fd);
23       break;
24       }
25     }
26   return (NULL);
27   }
28
```

File: FPRINTF.C

```
 1 #define NOCCARGC
 2 /*
 3 ** Yes, that is correct.  Although these functions use an
 4 ** argument count, they do not call functions which need one.
 5 */
 6 #include stdio.h
 7 /*
 8 ** fprintf(fd, ctlstring, arg, arg, ...) - Formatted print.
 9 ** Operates as described by Kernighan & Ritchie.
10 ** b, c, d, o, s, u, and x specifications are supported.
11 ** Note: b (binary) is a non-standard extension.
12 */
13 fprintf(argc) int argc; {
14   int *nxtarg;
15   nxtarg = CCARGC() + &argc;
16   return(_print(*(--nxtarg), --nxtarg));
17   }
18
19 /*
20 ** printf(ctlstring, arg, arg, ...) - Formatted print.
21 ** Operates as described by Kernighan & Ritchie.
22 ** b, c, d, o, s, u, and x specifications are supported.
23 ** Note: b (binary) is a non-standard extension.
24 */
25 printf(argc) int argc; {
26   return(_print(stdout, CCARGC() + &argc - 1));
27   }
28
29 /*
30 ** _print(fd, ctlstring, arg, arg, ...)
31 ** Called by fprintf() and printf().
32 */
33 _print(fd, nxtarg) int fd, *nxtarg; {
34   int  arg, left, pad, cc, len, maxchr, width;
35   char *ctl, *sptr, str[17];
36   cc = 0;
37   ctl = *nxtarg--;
38   while(*ctl) {
39     if(*ctl!='%') {fputc(*ctl++, fd); ++cc; continue;}
40     else ++ctl;
41     if(*ctl=='%') {fputc(*ctl++, fd); ++cc; continue;}
42     if(*ctl=='-') {left = 1; ++ctl;} else left = 0;
43     if(*ctl=='0') pad = '0'; else pad = ' ';
44     if(isdigit(*ctl)) {
45       width = atoi(ctl++);
46       while(isdigit(*ctl)) ++ctl;
47       }
48     else width = 0;
49     if(*ctl=='.') {
50       maxchr = atoi(++ctl);
51       while(isdigit(*ctl)) ++ctl;
52       }
53     else maxchr = 0;
54     arg = *nxtarg--;
55     sptr = str;
56     switch(*ctl++) {
57       case 'c': str[0] = arg; str[1] = NULL; break;
58       case 's': sptr = arg;        break;
59       case 'd': itoa(arg,str);     break;
60       case 'b': itoab(arg,str,2);  break;
61       case 'o': itoab(arg,str,8);  break;
62       case 'u': itoab(arg,str,10); break;
63       case 'x': itoab(arg,str,16); break;
64       default:  return (cc);
65       }
66     len = strlen(sptr);
67     if(maxchr && maxchr<len) len = maxchr;
68     if(width>len) width = width - len; else width = 0;
69     if(!left) while(width--) {fputc(pad,fd); ++cc;}
70     while(len--) {fputc(*sptr++,fd); ++cc; }
71     if(left) while(width--) {fputc(pad,fd); ++cc;}
72     }
73   return(cc);
74   }
75
```

File: FPUTC.C

```
 1 #define NOCCARGC  /* no arg count passing */
 2 #include stdio.h
 3 #include clib.def
 4 extern int _status[];
 5 /*
 6 ** Character-stream output of a character to fd.
 7 ** Entry: ch = Character to write.
 8 **        fd = File descriptor of perinent file.
 9 ** Returns character written on success, else EOF.
10 */
11 fputc(ch, fd) int ch, fd; {
12   switch(ch) {
13     case EOF:  _write(CPMEOF, fd); break;
```

(Continued on page 80)

Small-C Library (Listing Continued, text begins on page 50)

Listing Two

```
14      case '\n': _write(CR, fd); _write(LF, fd); break;
15      default:   _write(ch, fd);
16      }
17    if(_status[fd] & ERRBIT) return (EOF);
18    return (ch);
19    }
20  #asm
21  putc equ   fputc
22       entry putc
23  #endasm
```

File: FPUTS.C

```
 1  #define NOCCARGC  /* no arg count passing */
 2  #include stdio.h
 3  #include clib.def
 4  /*
 5  ** Write a string to fd.
 6  ** Entry: string = Pointer to null-terminated string.
 7  **        fd     = File descriptor of pertinent file.
 8  */
 9  fputs(string,fd) char *string; int fd; {
10    while(*string)
11      fputc(*string++,fd) ;
12    }
13
```

File: FREAD.C

```
 1  #define NOCCARGC  /* no argument count passing */
 2  #include clib.def
 3  extern int _status[];
 4  /*
 5  ** Item-stream read from fd.
 6  ** Entry: buf = address of target buffer
 7  **         sz = size of items in bytes
 8  **          n = number of items to read
 9  **         fd = file descriptor
10  ** Returns a count of the items actually read.
11  ** Use feof() and ferror() to determine file status.
12  */
13  fread(buf, sz, n, fd) char *buf; int sz, n, fd; {
14    return (read(fd, buf, n*sz));
15    }
16
17  /*
18  ** Binary-stream read from fd.
19  ** Entry:  fd = file descriptor
20  **        buf = address of target buffer
21  **          n = number of bytes to read
22  ** Returns a count of the bytes actually read.
23  ** Use feof() and ferror() to determine file status.
24  */
25  read(fd, buf, n) int fd, n; char *buf; {
26    char *cnt;  /* fake unsigned */
27    cnt = 0;
28    while(n--) {
29      *buf++ = _read(fd);
30      if(_status[fd] & (ERRBIT | EOFBIT)) break;
31      ++cnt;
32      }
33    return (cnt);
34    }
```

File: FREE.C

```
 1  #define NOCCARGC  /* no argument count passing */
 2  extern char *_memptr;
 3  /*
 4  ** free(ptr) - Free previously allocated memory block.
 5  ** Memory must be freed in the reverse order from which
 6  ** it was allocated.
 7  ** ptr    = Value returned by calloc() or malloc().
 8  ** Returns ptr if successful or NULL otherwise.
 9  */
10  free(ptr) char *ptr; {
11    return (_memptr = ptr);
12    }
13  #asm
14  cfree  equ    free
15         entry  cfree
16  #endasm
```

File: FREOPEN.C

```
 1  #define NOCCARGC  /* no argument count passing */
 2  #include stdio.h
 3  /*
 4  ** Close previously opened fd and reopen it.
 5  ** Entry: fn   = Null-terminated CP/M file name.
 6  **              May be prefixed by letter of drive.
 7  **              May be just CON:, RDR:, PUN:, or LST:.
 8  **        mode = "a"  - append
 9  **               "r"  - read
10  **               "w"  - write
11  **               "a+" - append update
12  **               "r+" - read   update
13  **               "w+" - write  update
14  **        fd   = File descriptor of pertinent file.
15  ** Returns the original fd on success, else NULL.
16  */
17  freopen(fn, mode, fd) char *fn, *mode; int fd; {
18    if(fclose(fd)) return (NULL);
19    if(_open(fn, mode, fd)==ERR) return (NULL);
20    return (fd);
21    }
```

(To be Continued in June Issue)

The Accent Finder

by Eddy Vasile

Eddy Vasile, 3314 Sawtelle Blvd. #28, Los Angeles, CA 90066.

If you have ever taken an introductory Spanish or French course, you probably know what accents are. You remember: those little marks that had to be placed on top of certain letters in certain words just to make your life bitter. French has three kinds of accents; the rules for their placement are complicated and have many exceptions. Spanish has only one accent and a single placement rule with fewer exceptions. Nonetheless, things can get complicated.

If you want to write Spanish correctly, you have three choices:

1. Learn the accent placement rule and all the exceptions (time factor: 7 days)
2. Write with a dictionary by your side and look up each word (time factor: 1 minute per word)
3. Write a program that tells the user where to put the accent and teaches the user the rules and exceptions (time factor: who cares, as long as I get to play?)

I wrote the Accent Finder program as an exercise in the use of sets in Pascal and also for the benefit of UCLA students interested in using the computer for humanities. Members of the Spanish department thoroughly tested the Accent Finder, and so far it has not produced any erroneous output. Should you find an error, please contact me for changes. The UCLA IBM4341 User's Group keeps a version of this program on the school's open access computer for any interested users.

A Brief Primer of Spanish Phonology

Believe it or not, there is an academy (in Spain, naturally) that supervises the use and development of Spanish. This institution has formulated rules for the placement of the accent and for the correct parsing of words into syllables (which is crucial for our algorithm).

Spanish orthography is almost a mapping of the phonetics (i.e., each letter is a symbol for a unique sound; you write what you hear). The phonetic inventory consists of the vowels (A, E, I, O, U) and a multitude of consonants more or less similar to English. There are also two liquids: L and R. These sounds are called liquids because of certain features (i.e., they can be held a long time, just like a vowel, and the sound-generating flow of air escapes on both sides of the tongue while it touches the palate). Vowels are strong (the set [A, E, O]) and nonstrong (the set [I, U]).

The Accent Placement Rule

For the purpose of placing accents, a Spanish word is described by two criteria:

1. The position of the syllable with the pronunciation emphasis
2. The quality of the last sound of the word

To decide whether an accent is needed, we apply each of the criteria to the word in question and compare the results.

Each criterion can mark a word with a plus or a minus. If the emphasis syllable is the penultimate one, then the word gets a plus; otherwise, the word gets a minus. If the word ends with a vowel, an N, or an S, then the word gets a plus from criterion 2; otherwise, the word gets a minus. If the word has two different signs (i.e., both plus and minus), then the word gets a written accent on the emphasis vowel. If the word has two identical signs, then no accent is required.

Let's look at an example. In the word *chicharon,* the syllables are:

1 CHI ** 2 CHA ** 3 RON

The emphasis is on syllable 3, not the penultimate one. Thus, criterion 1 marks the word with a minus. The last sound of the word is N. Thus, criterion 2 marks the word with a plus. The signs for *chicharon* are mixed, so you must place a written accent on the last syllable: *chicharón.*

Now let's look at *francesa.* The syllables are:

1 FRAN ** 2 CE ** 3 SA

The emphasis is on the penultimate syllable: the word gets a plus under criterion 1. The last sound is A, a vowel: the word gets another plus under criterion 2. The signs are the same, so no accent is required.

The Accent Placement Algorithm

The accent placement problem is now to find a way of coding the word description criteria (the emphasis position and the quality of the last sound). I chose to represent the criteria by means of two Booleans: CountPlus, which is true when the emphasis is on the penultimate syllable, and EndPlus, which is true when the word ends with an N, an S, or a vowel.

The setting of EndPlus is trivial. First, define the variable vowels of type SET OF CHAR:

```
Vowels := ['A', 'E', 'I', 'O', 'U']
```

Next, check to see if the last character of the word is in the vowel set. If it is, then EndPlus is set to true.

The setting of CountPlus is more complicated. The word must first be split into syllables (we'll cover that shortly), and the position of the emphasis must then be identified.

I chose to identify the position of the emphasis by asking the user. This accomplishes two goals: the user thinks about the rule and the pronunciation, and words such as *mama*, which can take the emphasis on different syllables according to the syntactic role, are covered.

The Algorithm for Syllable Parsing

Splitting a word into syllables is a process that relies on our linguistic intuitions. However, a fairly simple set of rules can be drafted for automatic parsing. We need three arrays of Booleans parallel with the string holding the word: Strong, Vowel, and Boundary. Strong is true if the sound in the string is a strong vowel. Vowel is true if the sound is a vowel. Boundary is true if the sound at the same position in the string is the boundary between two syllables.

All we must do now is to set the values of the elements of the Boundary array. Boundary is set to true (i.e., the respective sound is a border between two syllables) in the following cases:

1. If the current sound is T and the previous sound was N
2. If the current sound is a vowel and the previous sound was a consonant
3. If the current sound is a nonstrong vowel and the previous sound was a strong vowel
4. If both the current and the previous sounds are vowels

All this is done in the SetSylls procedure via a FOR loop, which checks the qualities of each sound and its neighbor's. Each time the qualifying conditions are met, the element of the Boundary array at that position is set to true.

In the SetSylls procedure, a final check is executed. If the procedure finds

the pattern vowel-consonant-consonant, then the Boundary is reset: the first consonant gets to be a boundary, and the second becomes part of the syllable initiated by the first. If the procedure encounters two neighboring consonants, then the second one marks a syllable boundary. One exception is LL, which in Spanish is pronounced like the English Y; this is coded in the procedure SplitConsonants, which checks for a consonant-consonant pattern (other than LL) by means of a FOR loop.

Two other consonant pairs should not be split: CH, which generates a single sound, just as its English counterpart, and liquids. The procedures WeldCH and WeldLiquids scan the CH and liquid patterns (using a FOR loop) and set the Boundary to true for the leading consonants.

Now that the syllable boundaries are set, we can concatenate all the letters marked false in the Boundary array to the letter marked true that precedes them, creating the bodies of the syllables that are kept in the Syllables array.

Program Output

The program displays the syllables with a delimiter between them and asks the user to pick the one with the emphasis. Next, the plus/minus rule is applied as described above; if the word has an accent, then it is displayed on the strong vowel of the syllable with the emphasis. The program also explains its decisions as it goes along, using the procedure InfoMsg.

The program provides for extensive error checking for entry of incorrect alphanumeric symbols and incorrect syllable numbers. Special procedures also are included for words that get accents according to syntactic context.

■■J

Accent Finder Listing

```
(*************************************************************************
The Accent program analyzes spanish words and determines if and where
there is a written accent. The program will ask for the word and then
it will display the word split in syllables and it will ask for the
number of the syllable carrying the pronounciation emphasis. The
syllables are split by stars.
If there should be a written accent, then its position will
be displayed.

The algorithm has considerable error checking with the use of sets
for  bad input or bogus syllable numbers.
                Author:         Eddy C. Vasile
                Last modified: Sept 15 83

NOTE: Please warn users that syllable parsing is sometimes
      disputed even in highly qualified linguistic circles.
      My parsing algorithm will produce a somewhat unorthodox
      syllable scheme when dealing with diphthongs, but only
      to suit the ultimate need of accent evaluation.
*************************************************************************)

program accent;
   const
      MaxChar = 50;       (*Maximum line size            *)
      MaxSyl  = 10;       (*Maximum number of syllables *)
      SylSize = 5;        (*Maximum size of a syllable  *)
      Radix   = 10;       (* base 10, used for conversions*)
   type
      CharSet       = set of char;
      WordType      = string[MaxChar];
      WordIndexType = 1..MaxChar;
   var
      Word     :WordType; (*The word to be processed*)
      Letters,            (*The alphabet             *)
      LowerCaseSet,       (*Lower case letters       *)
      Vowels,
      StrongVowels,
      Liquids :CharSet;
      Error:boolean;
```

```
(*************************************************************************
* function NUMERIC                                                       *
*   Given : 1-10 char token (InputStr)                                   *
*   Return : TRUE if it is a numeric constant.                           *
*            FALSE if it is not.                                         *
*   If any char is non-numeric, the token is non-numeric.                *
*************************************************************************)
function Numeric(InputStr : WordType): boolean;
   var
      i      : WordIndexType;        (* pos. chars of input string *)
   begin  (* Numeric *)
      Numeric := TRUE;                  (* return TRUE by default *)
      for i := 1 to length(InputStr) do          (* if any chars are *)
         if not (InputStr[i] in ['0'..'9']) (* non-numeric, *)
            then Numeric := FALSE           (* return FALSE *)
   end;  (* Numeric *)

(*************************************************************************
* function CONVERT                                                       *
*   Given : a string of digits (Instring).                               *
*   Return : its value (integer).                                        *
*   Convert from string to numeric by multiplying each digit by the      *
*     appropriate power of the specified base (Radix).                   *
*************************************************************************)
function Convert(Instring:WordType):integer;
   var
      Temp : Integer;
      i    : WordIndexType;          (* poses chars of input string *)
(*************************************************************************
* The DigValue function returns the numeric value of a single digit*
*************************************************************************)
function DigValue (c:char):integer;
   begin
      DigValue:=ord(c)-ord('0')
   end;

   begin  (* Convert *)
      Temp:=0;                          (* initially return value = 0 *)

      (* Beginning with left-most digit, multiply its value by the *
       *   specified base and add result to previous value.       *)
      if length(InString) > 0 then
         begin
            Temp:=DigValue(InString[i]);
            for i:=2 to length(InString) do
               Temp:=Temp * Radix + DigValue(InString[i])
         end (*if*)
  end;  (* Convert *)
(*********************************************
* The condense procedure preprocesses the input*
* word by compressing it and trimming both ends*
* If your compiler does not have these        *
* you may either ignore this proc or write the *
* functions your self.                        *
*********************************************)
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
function Condense(Instring:WordType):WordType;
   begin
      Condense:=compress(ltrim(trim(Instring)))
   end; (*Condense*)

(************************************************
*The upper case procedure uppercases a line of  *
*input. If your machine is ASCII then this is   *
*fine, if EBCDIC then you must change the       *
*conversion factor to 64 and add                *
************************************************)
function Uppercase(InputStr : WordType): WordType;
var
   i              : WordIndexType;
begin
   for i := 1 to length(InputStr) do
      if  (InputStr[i] in LowerCaseSet)
      then InputStr[i] := chr(ord(InputStr[i])-32);
   Uppercase := InputStr
end; (* Uppercase *)

(*******************************************************
*procedure CheckChars checks for the validity of the   *
*input by identifying garbage characters               *
*******************************************************)
procedure CheckChars(var Error:boolean; word:wordType);
   var
      i:WordIndexType;
   begin
      error:=false;
      for i:=1 to length(word) do
          if not (word[i] in Letters)
          then error:=true
   end; (*CheckChars*)

(*****************************************************
*procedure MImsg offers a message if the word is MI  *
*Special treatment for two letter words is necessary *
*due to the number of pecularities.                  *
*****************************************************)
procedure MImsg;
   begin
      writeln('Mi does not take an accent if it is used as a',
              ' possesive pronoun,');
      writeln('but it takes an accent when used as a reflexive',
              ' pronoun .');
      writeln('Example:Mi perro es feo');
      writeln('Me afeito a mi'' mismo.')
   end;


(*****************************************************
*procedure SImsg offers a message if the word is SI  *
*****************************************************)
```

```
procedure SImsg;
   begin
      writeln('The word SI has an accent when used as ''yes'',');
      writeln('or when used as a reflexive pronoun.');
      writeln('Example:! Si'', me duelen los dientes!');
      writeln('Se afeita a si'' mismo.')
   end;


(*******************************************************
*procedure ELmsg offers a message if the word is EL   *
*******************************************************)
procedure ELmsg;
   begin
      writeln('El has an accent when used as a pronoun and not');
      writeln('when used as an article.');
      writeln('E''l me dio un puno');
      writeln('!Deme el dinero!')
   end;
(*************************************************************
The prompt procedure will prompt user for a word, scan it
for invalid input and set an error parameter to true if
so. Else, the word is passed to the PROCESS procedure.
*************************************************************)
procedure prompt(var word:WordType; var Error:boolean);
   begin
      write('Enter the word > ');
      readln(word);
      word:=UpperCase(Condense(word))
    end; (*prompt*)


(*************************************************************
*The master procedure of the program that processes the     *
*word by dividing it in syllables and computing whether     *
*the word has or does not have an accent                    *
*************************************************************)
procedure process(word:wordType);
   type
      LetPosType    = 1 .. MaxChar;
      SylPosType    = 1.. MaxSyl;
      SylArray      = array[1..maxSyl] of string[SylSize];
      SoundType     = array[1..MaxChar] of boolean;
   var
      TempString: WordType;
      Syllables:SylArray;
      Vowel,
      Strong,
      Boundary        : SoundType;
      EndPlus,
      CountPlus       : boolean;
      LastSyl,
      StressSyl       : SylPosType;
      StressLet       : LetPosType;
```

(continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
(*********************************************
*The CheckNumeric procedure verifies validity*
*of numeric input in order to proevent a     *
*program crash on bad input                  *
*********************************************)
procedure CheckNumeric(var TempString:WordType);
   begin
     while not Numeric(TempString) do
        begin
           writeln('**** Garbage Input ***');
           write('Please enter a numeric value > ');
           readln(TempString)
        end
    end; (*CheckNumeric*)

(*********************************************
*procedure CheckSyl verifies the validity    *
*of the syllable number that is supposed     *
*to carry the stress.                        *
*Ie. It should not be 0 and not greater      *
*then the number of syllables                *
*********************************************)
procedure CheckSyl( var StressSyl:integer);
   begin
     while not (StressSyl in [1..LastSyl]) do
        begin
           writeln('Invalid syllable number');
           write('Enter correct emphasis syllable >');
           readln(TempString);
           CheckNumeric(TempString);
           StressSyl:=Convert(TempString)
        end
   end; (*CheckSyl*)


(*********************************************************
*The Initialize procedure sets the variables to initial*
*values in preparation for a new word of input          *
*********************************************************)
procedure Initialize(var Strong,Boundary,vowel:SoundType;
                     var Syllable:SylArray);
   var
      i:integer;
   begin
      for i:=1 to MaxChar do
         begin
            boundary[i]:=false;
            vowel[i]:=false;
            Strong[i]:=false
         end;
      boundary[length(word)+1]:=true;
```

```
      for i:=1 to MaxSyl do Syllable[i]:=''
   end; (*Initialize*)

(************************************************
*procedure SetQuality sets the quality of each   *
*sound of the word. Ie. if a vowel then the vowel*
*flag is turned on and if a strong vowel then the*
*strong flag is turned.                          *
************************************************)
procedure SetQuality(var Strong,vowel:SoundType);
   var
      i:integer;
   begin
      for i:=1 to length(word) do
         if word[i] in vowels
         then
            begin
               vowel[i]:=true;
               if word[i] in StrongVowels
               then Strong[i]:=true
            end
   end; (*SetQuality*)

(******************************************************
*the SetSylls procedure sets the boundary between the  *
*syllables of the word according to strength and value.*
******************************************************)
procedure SetSylls(var Boundary:SoundType; word:wordType;
                       vowel   :SoundType);
   var
      i:integer;
   begin
      for i:=2 to length(word) do
         begin
            if ((not vowel[i]) and vowel[i-1])
            or ((word[i-1] = 'N') and (word[i]='T'))
            or (strong[i-1] and (not strong[i]))
            or (strong[i-1] and strong[i])
            or (strong[i] and (not strong[i-1]) and vowel[i-1])
            then boundary[i]:=true;
            if i>2 then
            if (vowel[i-2] and (not vowel[i-1]) and (not vowel[i])
                and (word[i]=word[i-1]))
            then
               begin
                  boundary[i-1]:=true;
                  boundary[i]:=false
               end
         end
   end; (*SetSylls*)

(************************************************************
*The SplitConsonants splits two neighbouring consonants in   *
*syllables (except ll)                                       *
************************************************************)
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
procedure SplitConsonants(var boundary:SoundType; word:WordType;
                              vowel   :SoundType);
   var
      i:integer;
   begin
      for i:=3 to length(word) do
         if vowel[i-2] and (not vowel[i-1]) and (not vowel[i])
         then
            begin
               boundary[i-1]:=false;
               boundary[i]:=true
            end
   end; (*SplitConsonants*)
(************************************************
*The WeldCH procedure insures that C and H go in  *
*the same syllable                                *
************************************************)
procedure WeldCH(var boundary: SoundType; word:wordType;
                     vowel   :SoundType);
   var
      i:integer;
   begin
      for i:=3 to length(word) do
         begin
            if (word[i-1]='C') and (not Boundary[i-1]) and
               (word[i]='H') and boundary[i]
            then
               begin
                  Boundary[i-1]:=true;
                  Boundary[i]:=false
               end
         end
   end; (*WeldCH*)
(************************************************
*Liquids (l and r) must be spliced to the next    *
*syllable                                         *
************************************************)
procedure WeldLiquids(var boundary: Soundtype; word:wordType;
                          vowel   : SoundType);
   var
      i:integer;
   begin
      for i:=3 to length(word) do
         if vowel[i-2] and
            (not vowel[i-1]) and (not vowel[i]) and
            (word[i] in liquids)
         then
            begin
               boundary[i-1]:=true;
               boundary[i]:=false
            end
  end; (*WeldLiquids*)
```

```
(*************************************************************
*The divide procedure sets the array of strings 'Syllables'*
*to the values of the word syllables by slicing together   *
*the letters of a word up to a letter marked as a syllable *
*************************************************************)

procedure divide(boundary:SoundType; var Syllables:SylArray);
  var
     i,
     j:integer;
  begin
     i:=1;
     Syllables[1]:=copy(word,1,1);
     j:=2;
     while j<=length(word) do
       begin
          if not boundary[j]
          then Syllables[i]:=Syllables[i]+copy(word,j,1)
          else
             begin
                Syllables[i+1]:=copy(word,j,1);
                i:=i+1
             end;
          j:=j+1
       end
   end; (*divide*)

(*************************************************************
*The EvalSylls procedure displays the word split in syllables*
*and it also returns their total number.                     *
*The syllables are numbered.                                 *
*************************************************************)
procedure EvalSylls(syllables: SylArray; var LastSyl:integer);
   var
      i:integer;
   begin
      i:=1;
      while syllables[i]<>'' do
         begin
            write('***',i:1,'->',syllables[i]);
            i:=i+1
         end;
      LastSyl:=i-1
   end; (*EvalSyll*)


(*************************************************
*The procedure SetStressLet will set the position  *
* of the stress letter by doing some calculations  *
* I.e. looking for the stressed vowel within the   *
* stressed syllable                                *
*************************************************)
procedure SetStressLet(var StressLet:integer);
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
    var
       SyllCount,
       i          :integer;
    begin
       SyllCount:=1;
       for i:=1 to length(word) do
          begin
             if Boundary[i]
             then SyllCount:=SyllCount+1;
             if (SyllCount=StressSyl) and Vowel[i]
             then StressLet:=i
          end
    end; (*SetStressLet*)

(*********************************************
*procedure DisplayAccent displays the word    *
*with its written accent as a slash next to   *
*the letter supposed to carry it.             *
*********************************************)
procedure DisplayAccent(word:WordType);
    var
       i:integer;
    begin
       writeln('The word ',word,' has a written accent ');
       for i:=1 to length(word) do
          begin
             write(' ',word[i]);
             if i=StressLet then write('/')
             else write('  ')
          end;
       writeln
    end; (*DisplayAccent*)

(*********************************************
procedure InfoMsg comes to work after all    *
procedures have done their work and it       *
explains why certain decisions were made     *
*********************************************)

procedure InfoMsg(word: WordType; EndPlus, CountPlus: boolean);
    begin
       writeln('The final letter is ',word[length(word)]);
       write('Thus a ');
       if EndPlus then write('+')
       else write('-');
       writeln(' must be assigned for the word ending.');
       writeln('The emphasis is on syllable #',StressSyl:1);
       write('Thus we must assign ');
       if CountPlus then write('+')
       else write('-');
       writeln(' for the emphasis placement.');
       writeln('Reasons considered for the above decision: ');
       If EndPlus<>CountPlus
       then writeln('The signs are different so there is an accent!')
       else writeln('The signs are the same so there is no accent!');
       if (not strong[length(word)-1]) and vowel[length(word)-1] and
```

```
        Strong[Length(word)]
      then writeln('The word ends with a diphthong, there is an accent.')
      if strong[StressLet-1] and (not strong[StressLet])
      then writeln('A split diphtong, there is an accent.');
      if (word[length(word)-1]='P') and (word[length(word)]='S')
      then writeln('Words ending in PS always get accents!');
      if (length(word)=3) and (not vowel[1])
      then writeln('Three letter words are affected by semantics',
                   ' (Ex. mas vs. ma's)')
   end; (*InfoMsg*)

begin(*of the process procedure*)
   Initialize(Strong,Boundary,vowel,Syllables);
   SetQuality(strong,vowel);
   SetSylls(boundary,word,vowel);
   SplitConsonants(boundary,word,vowel);
   WeldCH(boundary,word,vowel);
   WeldLiquids(boundary,word,vowel);
   if not vowel[length(word)]
   then boundary[length(word)]:=false;
   divide(boundary,syllables);
   EvalSylls(syllables,LastSyl);
   write('Enter the number of emphasis syllable > ');
   readln(TempString);
   CheckNumeric(TempString);
   StressSyl:=Convert(TempString);
   CheckSyl(StressSyl);
   SetStressLet(StressLet);
   EndPlus:=false;
   CountPlus:=false;
   if not ((not vowel[length(word)]) and (not vowel[length(word)-1]))
   then if (vowel[length(word)]) or
           (word[length(word)] in ['N','S'])
        then EndPlus:=true;
   if LastSyl-StressSyl=1 then CountPlus:=true;
   if (EndPlus<>CountPlus) or
      ((length(word) > 3) and
      strong[StressLet-1] and (not strong[StressLet])) or
      (((not strong[length(word)-1]) and Vowel[length(word)-1]) and
      Strong[length(word)])
   then DisplayAccent(word)
   else writeln('No accent !');
   InfoMsg(word,EndPlus,CountPlus)
end;  (*Process*)

begin (*main*)
   word:='xxx';
   vowels:=['A','E','I','O','U'];
   liquids:=['L','R'];
   StrongVowels:=['A','E','O'];
   LowerCaseSet:=['a'..'z'];
```

(Continued on next page)

Accent Finder Listing (Listing Continued, text begins on page 82)

```
Letters:=['A'..'Z'];
writeln('Please remeber that all interrogatives get an accent');
writeln('when used in a question.');
writeln('Ex: ?Que'' dice?');
writeln('    E''l dice que tiene once anos.');
writeln('To stop enter ''Return'' without input');
while word<>'' do
   begin
      Prompt(word,error);
      if (word<>'') and (not error)
      then
         begin
            if length(word) < 3 then
            case length(word) of
               1: writeln('Monosounds do not have accents');
               2: if word='SI' then SImsg
                  else if word='MI' then MImsg
                       else if word='EL' then ELmsg
                            else
                               writeln('Probably no accent !')
            end
            else Process(word)
         end
      end;
   writeln('!!Que la fuerza este'' contigo!!')
end.
```

End Listing

Solutions to Quirks in dBASE II

At Money Tree Software in Corvallis, Oregon, we offer a service to financial planners called MoneyNet. MoneyNet is a computer data base accessed by a modem from which planners can retrieve timely data about various financial matters.

The entire system is written in dBASE II. We tried several of the dBASE II look-alikes, which claim more records, more fields, and more memory variables, before committing to dBASE II. Let me say, with firsthand experience, that from a high-level language perspective dBASE II is the clear choice of most serious programmers.

However, as good as dBASE II is, it does have some quirks.

The quirk I will address here is the lack of an INKEY function like that found in other high-level languages. The INKEY can be used to interrupt a long listing or print-out without aborting the program *entirely.*

The Problem

Almost any standard CP/M text file can be converted to a dBASE II data file simply by creating a data base of one field, character type, and length of 80 then appending from the desired text file using the SDF option. Each record of the data base will be one line of text. Displaying the text can be a function of a normal DO-LOOP like this:

```
USE TEXTFILE
DO WHILE .NOT. EOF
? LINE
SKIP
ENDDO
```

But suppose this routine is part of a menu-driven, end-user application and the user decides in the middle of the DO-LOOP that he or she wants to stop reading this text and return to the main menu. How can the user abort the loop from the keyboard? ESCAPE has been turned off, since we don't want the end user at the DOT-PROMPT level, and it wouldn't meet the user's need anyway since he or she wants program flow to continue after exiting the DO-LOOP.

by Gene Head

Gene Head, 2860 NW Skyline Dr., Corvallis, OR 97330.

We need a way to check whether a key has been pressed at the keyboard after each record (one line of text) has been displayed. That way, our DO-LOOP could be aborted with a control-C from the keyboard as shown in Figure 1 (page 99).

The standard answer is to write a CALLable machine language routine to monitor the keyboard, usually by doing a BIOS function call. Unfortunately, this scheme will not work because dBASE II polls the keyboard at least once after a character is sent to the console, gobbling up the keyed-in character before any machine language call can grab it. No matter where in the program we put our call to console I/O check, it is very unlikely that our subroutine will recognize the user's key press before dBASE II intercepts it.

A Workable Solution

A disassembly examination of the dBASE II program shows that console input goes something like Figure 2 (page 99). It looks like the solution could be a simple matter of PEEKing at SAVEIT to see the last key pressed. Unfortunately, dBASE II zeros this byte after it uses the keyed-in data. This routine is called *very often*, and SAVEIT will be zero most of the time, containing the keyed-in character for only a very short period. My modification saves the keyed-in byte to a safe RAM location undisturbed by dBASE II (Figure 3, page 99).

This modification means that SAVEIT continues to perform as expected, but LASTKEY is filled with the keyed-in character and remains unchanged until another key is pressed or it is changed with a POKE command. Now we can monitor this location, much like BASIC allows for the INKEY command. The DO-LOOP will print text until the user inputs a control-C (or any desired character) from the keyboard or the end of file is found (Figure 4, page 99).

Of course, it isn't quite as easy as it sounds because there is no room to do a

Solutions to dBASE II Listing

```
A>DDT DBASE.COM
DDT VERS 2.2
NEXT PC
4E00 0100
-L 3A81
   3A81    MVI   E,FF
   3A83    MVI   C,06
   3A85    CALL  0005
   3A88    ORA   A
   3A89    RZ
   3A8A    STA   4378    <--- PATCH JMP HERE
   3A8D    RET
   3A8E    PUSH  B
   3A8F    PUSH  D
   3A90    PUSH  H
   3A91    CALL  3A62
-
-A3A8A
3A8A  JMP  14A          <--- PATCH JMP HERE
3A8D
-
-A14A
014A  STA  4378         <--- INSTRUCTION WE PATCHED OUT
014D  STA  151          <--- ADDED INSTRUCTION
0150  RET
0151  DB  0             <--- 337 DECIMAL (LASTKEY)
0152
-
-GO
A>SAVE 77 DBASE.COM
A>
```

End Listing

```
USE TEXTFILE
DO WHILE .NOT. EOF .AND. KEY:PRESS<>3          * 3=CONTROL-3
? LINE
SKIP
ENDDO
```

Figure 1.

```
CONIN   MVI     E,OFFH    ; SET UP FOR DIRECT CONSOLE INPUT
        MVI     C,9
        CALL    BIOS      ; DO DIRECT CONSOLE I/O
        ORA     A
        RZ                ; NOTHING TYPED SO RETURN
        STA     SAVEIT    ; SAVE TYPED KEY
        RET
```

Figure 2.

```
CONIN   MVI     E,OFFH    ; SET UP FOR DIRECT CONSOLE INPUT
        MVI     C,9
        CALL    BIOS      ; DO DIRECT CONSOLE I/O
        ORA     A
        RZ
        STA     SAVEIT    ; SAVE keyed-in character
        STA     LASTKEY   ; . . . and to a SAFE location
        RET
```

Figure 3.

```
USE TEXTFILE
DO WHILE .NOT. EOF .AND. PEEK(LASTKEY) <> 3   * 3=CONTROL-C
? LINE
SKIP
ENDDO
```

Figure 4.

```
? 'YOU HAVE FIVE SECONDS TO ANSWER  -->  '
STORE 100 TO TIMER
POKE 0,337
DO WHILE TIMER .AND. PEEK (337) = 0
STORE TIMER -1 TO TIMER
STORE PEEK (337) TO ANSWERED
ENDDO
IF ANSWERED
? 'YOU ENTERED AN ANSWER IN TIME'
ELSE
? 'YOU FAILED TO ENTER AN ANSWER IN
ENDIF
```

Figure 5.

direct in-line patch in dBASE II and my modification adds three bytes of code. So we have to do some redirection to an unused RAM area in dBASE II.

The screen I/O, as defined using the INSTALL.COM program, is saved in low memory from about 10D hex to about 152 hex. Because the codes starting at 14A hex are all zeros in my installed dBASE II, this is where I put the patch. As far as I can tell, dBASE II never addresses this RAM. *Your installed dBASE II may use these bytes*, so first look around this area for eight consecutive bytes of zeros and then put your patch there.

This is what will happen. Instead of storing the keyed-in character at SAVEIT using the STA SAVEIT instruction, we use these three bytes to do a jump to our unused area of RAM in the screen data table. Here we do the first store just like the store we patched over: STA SAVEIT. In addition, we save the keyed-in data to a safe RAM location with STA LASTKEY. Finally, we return. Our patch is done and dBASE II will not even know it is there!

If you use dBASE II 2.4, the DDT session in the Listing (page 98) will permanently modify your dBASE II to recognize a keyed-in character by PEEK (337). Now a PEEK (337) will always show the last key pressed. This can be used to time-out input, as shown in Figure 5 (below).

The DO-LOOP application has already been explained, and I hope you can come up with unusual applications I haven't thought of.

Next time I will present a short piece on verifying zip codes in mailing lists.

■■J

OF INTEREST

by Michael Wiesenberg

But Does It Have "Of Interest"?

Want to find all articles about your IMSAI, Sorcerer, or HP 87? The **Microindex**, from Serious Personal Computing, is various flavors of periodicals that index the multiplicity of articles in over 70 computer magazines, a sort of Readers' Guide to Periodical Literature of computer publications. *Microindex*, a monthly for libraries, universities, and large businesses, cross-references the entire spectrum of computer magazines, and costs $99 a year or $12 an issue. The monthly *Abridged Microindex*, for smaller organizations, indexes a subset of 32, at $49 per year or $6 per issue. Yes, *DDJ* is in it. Annual *Microindexes* specific to various magazines cost $5 to $12 each. I know that you will all want to order the *Microindex* to *Dr. Dobb's Journal*, Volume 8, for $7. Also available are complete one-volume monthlies and annuals on various topics, as education, business, Apple, IBM, Commodore, and so on.

C You on 8086

Version 2 of **Lattice C** is now available for 8086 and 8088 for most MS-DOS computers. Programs can address 1Mb of memory, and be as long as 1Mb. The new library handles multilevel file directories and is compatible with recent versions of Unix. Compiler costs $500, and library source is also $500. You can get an update from whoever supplied your original.

Million Dollar Computer Poker Challenge

Can a computer play intelligent poker? You wouldn't think so to judge from the poker games available on today's micros, but **Mike Caro** ("The Mad Genius of Poker"), renowned poker writer and games programmer, is so convinced his Apple can beat any human that he has issued a $100,000 challenge to several world-class poker players, and been accepted by two former champions of **The World Series of Poker**. A Vegas casino owner has offered a counterchallenge: to play for *one million dollars cash*. To ensure that the computer is not "helped" in its decisions by the operator, special decks of cards with optical codes on the faces and optical scanners will be provided by the **Data Bar Corporation**. Think you can play poker better than a computer? You might get a chance to try if you've got a million dollars to back you up. Better yet, can you program your computer to beat Caro's? Win or lose, the publicity for your game would be worth well over a million dollars.

Caro also plans on releasing the challenge soon as a home game, with the slogan, "Play the program that won a million dollars in the World Series of Poker." I've seen an advance copy, and the card graphics are impressive, much better than those tiny attempts at exact representation that are impossible to read that I've seen in other commercial games.

But What if Your Printing Isn't Legible?

For those who get tired of using keyboards, Pencept's **Penpad** lets you input data by handwriting. I mean hand *printing*, and you'll have to learn to represent lowercase letters with half-height capitals, and you have to stay within the lines just like you did in first grade. The real value in this product seems to be in designing custom forms, in conjunction with the **Penpatch** utility.

Penpad is directly compatible with VisiCalc, MultiPlan, Lotus 1-2-3, and WordStar. For $995 on the PC, you get tablet, ballpoint with side-mounted activation button, and a board for the PC. The board has its own MC68K CPU that has the character recognition software, sending input that looks to the computer like it came from the keyboard. Penpad is also what they call "the mouse that writes," because stylus motion and button pressing can be translated into cursor commands.

Save A Sorcerer

Now that Exidy no longer makes them, how do you thousands of Sorcerer owners get service and answers to your questions? Contact **Jack MacGrath**, who provides technical support, has parts, services Micropolis drives, and buys and sells systems.

Less for 40

You may recall my predicting a year ago that prices would come down considerably on hard disks with increased storage. CompuPro's **H40 Hard Disk Subsystem** for its IEEE 696/S-100 bus compatible microcomputers costs $5495, and comes with a 40Mb Quantum 5.25-inch hard disk, CompuPro's Disk 3 DMA disk controller, CP/M-80 or -86, and a double-density Qume Trak 842 floppy disk drive with 2.8Mb for backup on single- or double-density, single- or double-sided media.

And, while they've got our attention, CompuPro would like us to know that they have the first commercially available IEEE 696-compatible, 80286-based, multiuser microcomputer. This is, of course, Intel's new CPU that is upward compatible from the 8088 and 8086, but runs at 6 MHz, or twice as fast as 8086- or 68K-based systems. The **System 816/F** also has an 80287 math coprocessor. It comes with 512K 16-bit static memory, expandable to 1Mb, 12 serial ports, one Centronics-compatible printer port, one parallel port, 1.2Mb floppy storage, a 40Mb hard disk, and CP/M-86 and MP/M-86. $14,995 walks away with it.

Or Pascal

Limbic Systems has a **Pascal Compiler** for the Commodore 64 that generates native code. The package comes with linker, debugging facilities, and editor, and costs a phenomenal $50.

Zenith Pretends It's PC

Z-Util from Lindley Systems is IBM PC emulation for the Zenith Z-100, emulating keyboard input, screen printing and scrolling, and printer output, so that WordStar, the Perfect series, and dBaseII all run. Only modem and graphics programs don't run, because of the differences in hardware. Screen dumps, however, do work, with automatic logic-seeking and correct screen aspect ratios. All this for $35.

New Character for Old Printwheels

How often have you wished for an odd symbol on your printer: an at-sign, or copyright symbol, for instance, or maybe that elusive ™ that actually exists on some printwheels? **Business Support Services** will replace any character on your Diablo, Xerox, or Qume metal or plastic printwheel with any other available character.

Catch Your Words

You can get an inexpensive printout basket on which your printer sits from **Systems Enhancement Engineering**. $22.50 for the 12-inch model and $24.50 for 18, plus $3 p. and h.

Not Another Cute Head!

Tweaker, from Kludge Enterprises, is a hardware/software product for programmers and hardware engineers that puts that extra little bit into products always needed to ship them out the door. Specify format. $595 FOB VARPTR LAX.

Contact Points

Business Support Services, Inc., 705 Butternut Ave., Royal Oak, MI 48073; (313) 585-4736.

Caro's Poker Challenge: contact Henri Bollinger Agency, 9200 West Sunset Boulevard, Los Angeles, CA 90038; (213) 274-8483.

CompuPro, 3506 Breakwater Court, Hayward, CA 94545; (415) 786-0909.

Data Bar Corporation, 10202 Crosstown Circle, Eden Prairie, MN 55344; (800) 672-2776.

Lattice, Inc., Box 3072, Glen Ellyn, IL 60138; (312) 858-7950.

Limbic Systems, 560 San Antonio Road, Suite 202, Palo Alto, CA 94306; (415) 424-0168.

Lindley Systems, 21 Hancock Street, Bedford, MA 01730; (617) 275-6821 (evenings and weekends).

Jack MacGrath, 73 Jordan Road, North Chelmsford, MA 01863; (617) 251-4776.

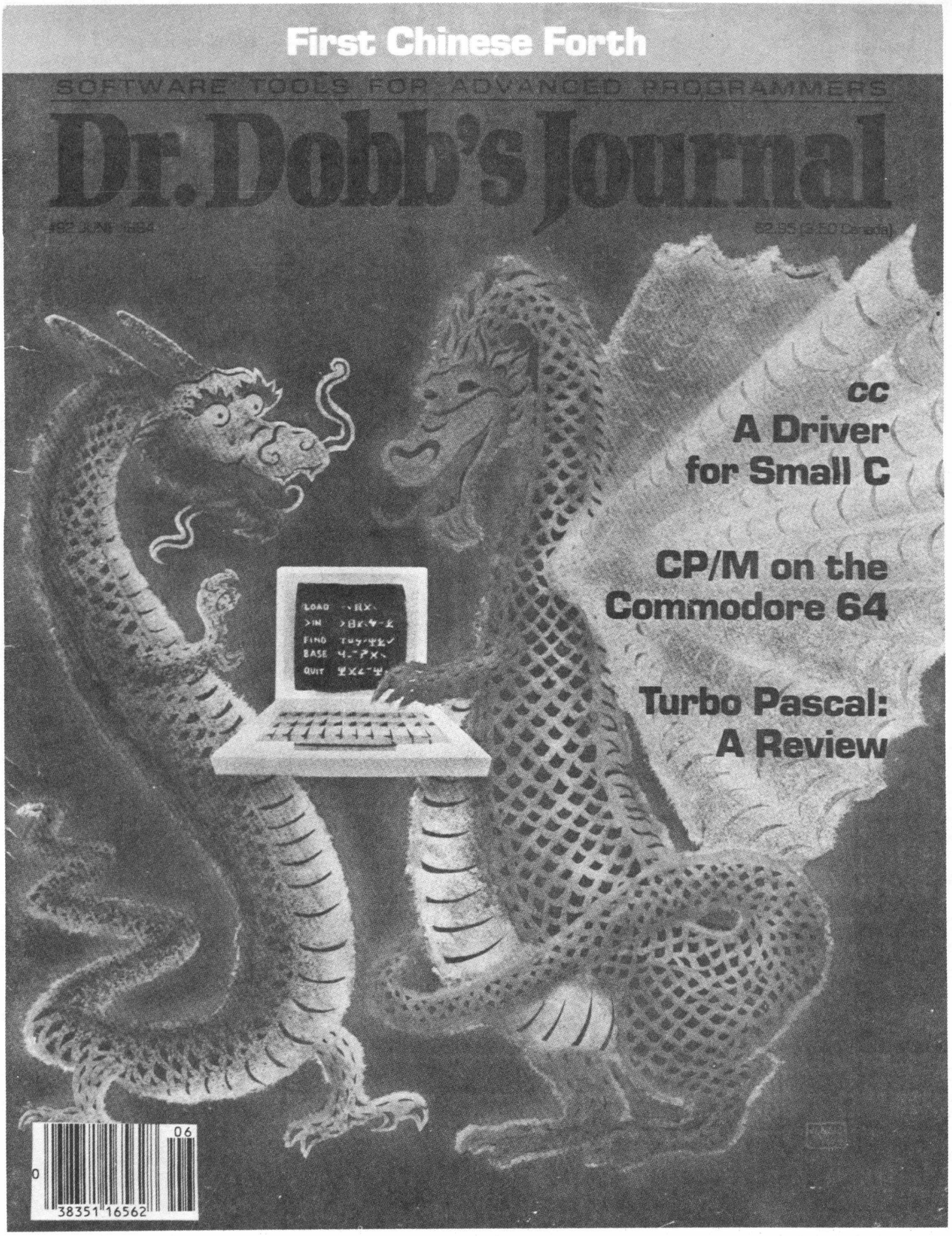
Pencept, Inc., 39 Green St., Waltham, MA 02154; (617) 893-6390.

Serious Personal Computing, Box 7059, South Nashua, NH 03060; (603) 888-1376.

Systems Enhancement Engineering, Box 40215, Indianapolis, IN 46240; (317) 844-8817.

Unified Software Systems, Box 2644, New Carrollton, MD 20784; (301) 552-9590.

■■J


First Chinese Forth
SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal
#82 JUNE 1984
$2.95 ($3.50 Canada)
cc
A Driver
for Small C
CP/M on the
Commodore 64
Turbo Pascal:
A Review
LOAD
>IN
FIND
BASE
QUIT
0 38351 16562 06

Dr. Dobb's Journal
Editor-in Chief: *Michael Swaine*
Editor: *Reynold Wiggins*
Managing Editor: *Randy Sutherland*
Contributing Editors:
Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg
Copy Editors: *Polly Koch, Cindy Martin*
Typesetter: *Jean Aring*

Circulation and Promotions Director: *Beatrice Blatteis*
Fulfillment Manager: *Stephanie Barber*
Direct Response Coordinator: *Maureen Snee*
Promotions Coordinator: *Jane Sharninghouse*
Single Copy Sales Coordinator: *Sally Brenton*
Single Copy Sales: *Lorraine McLaughlin*
Circulation Assistant: *Kathleen Boyd*

Advertising Director: *Susan Cohen Strange*
Advertising Sales: *Alice Hinton, Walter Andrzejewski*

Design Director: *Fred Fehlau*
Production/Art Director: *Shelley Rae Doeden*
Production Manager: *Detta Penna*
Production: *Alida Hinton*
Cover Illustration: *Janaia Donaldson*

M&T Publishing, Inc.
Publisher and Chairman of the Board: *Otmar Weber*
Director: *C. F. von Quadt*
President: *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points. Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R. B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. – Japan, Computer Services – Australia, Computer Store – New Zealand, Computercollectief – Nederland, Homecomputer Vertriebs GMBH – West Germany, International Presse – West Germany, La Nacelle Bookstore – France, McGill's News Agency PTY LTD – Australia, Progresco – France.

People's Computer Company
Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, Box E, Menlo Park, CA 94026, a non-profit, educational corporation.

In This Issue

Over the next few months you will be seeing some cosmetic changes in *Dr. Dobb's*. Some will be intended to enhance newsstand visibility, some to enhance general readability, some to increase contact or involvement with the readership. You may have noticed when you picked up this issue that the logo is slightly different, as is the descriptor line above the title. In future issues you will see some variation in format for various sections of the magazine, and our conversion in July to a new typesetting system will result in greater readability as well. This issue we have added the section you are presently reading.

With the Editorial page being exclusively devoted to essays each month, we found that we needed some space in which to let you know about things of importance regarding the magazine. This is that space. You will find highlights of the current issue and coming topics, announcements of events or other items of interest, acknowledgements of special contributions to the magazine, or other things we consider noteworthy.

Since we are still settling in, we will postpone details on our growing number of projects, including the bulletin board to which we have alluded, until they progress a bit further. One project that is in full swing, however, is our Fifth Generation Programming Competition. If you did not notice our full-page announcement last month, take a look at page 81 this month for details. In addition, our efforts to improve services should be evident from the growth of the masthead. Many of the new names you will find there are folks who will be making sure that *DDJ* is more available on the newsstands and that subscriptions are serviced better.

The contents page (facing) should give you a good overview of what we have for you this month. During the next few months you can look forward to continued coverage of C and Small C, including a preprocessor for the Small C compiler. Forth will remain an active topic, with our annual Forth issue scheduled for September. Some other items to watch for include an infinite key cryptography system, discussion of Resident System Extensions under CP/M Plus, and some articles on mathematical computation.

Reynold Wiggins

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to their remarks to the editors concerning accuracy and relevance of manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness.

The board includes nearly fifty experts from diverse areas of the computer industry and the academic community. Because of space considerations, we can print a list of the entire board only a few times each year. Monthly, however, we do print the names of the referees who contributed their insights on material in that particular issue. Your humble editors, who bear the burden of choosing how material ultimately appears, are grateful for the beneficial insights we receive.

The referees who contributed to this month's issue are:

David E. Cortesi, Contributing Editor, *DDJ*

Kim Harris, Forthright Enterprises

J. E. Hendrix, Office of Computing & Information Services, University of Mississippi

William Ragsdale, President, Dorado Systems

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

June 1984
Volume 9, Issue 6

CONTENTS

ARTICLES

14 CP/M on the Commodore 64 — Including Two BIOS Modifications
by Walter G. Piotrowski
This article presents a brief overview of CP/M on the Commodore 64 along with details of how to patch the BIOS to accommodate two 1541 disk units and/or an RS-232 printer

28 dBase II Programming Techniques
by Gene Head
Departing from last month's discussion of dBase quirks, the author presents a machine-language sub-routine designed for mailing list applications. The technique allows one to partially verify Zip codes and validate states.

32 First Chinese Forth: A Double-Headed Approach
by Timothy Huang
Challenged by the task of providing a computer language which could be used by Chinese-speaking people, this author chose Forth as the most logical computer language to implement in Chinese.

40 *cc* — A Driver for a Small-C Programming System
by Axel T. Schreiner
Patterned after the Unix *cc* command, this utility accepts options and file specifications, then uses CP/M's SUBMIT facility to arrange the proper amount of preprocessing, compiling assembly, and loading.

56 A New Library For Small C (Part II)
by James Hendrix and Ernest Payne
The listing of the new library for Small C v.2, begun last month, concludes this issue with the second half of the user functions.

70 Comments on Sixth Generation Computers
by Michael J. Doherty
The dialogue continues: Doherty responds to Grigonis' essay last month on sixth generation computers.

DEPARTMENTS

9 Editorial

10 Letters

12 Dr. Dobb's Clinic
Quasi-Disk; Fooling RMAC; DIR Not-Quite-So-Full; Without Fear or Good Taste

74 Software Reviews
Turbo Pascal

82 16-Bit Software Toolbox
Professional Basic; The Filepath Utility; Beating the Laws of Thermodynamics; Floating-Point Benchmarks; More on the Microsoft 8086 Assembler; A Toolbox Installment (word_to_hex and byte_to_hex)

86 C/Unix Programmer's Notebook
8086 memory models; Long pointer package for C compilers

108 Of Interest

EDITORIAL

About those dragons.

Dr. Dobb's Journal prides itself on providing useful tools and information for advanced programmers. Last month we ranged beyond the fields we know and published an article on "sixth generation computers," whose author explored the implications of applying bizarre twists of quantum physics to computer design. The discussion of faster-than-light particles with imaginary rest mass will probably never help anyone write a better spreadsheet program or word processor; it frankly bordered on fantasy. Here's why we published it.

China story, version 1.0: China, until the twentieth century, was largely isolated from Western culture. Homage to traditions had kept some strange practices (acupuncture, footbinding) in force while modern science developed in the West. When, in the twentieth century a new generation of Chinese, unwilling to suffer longer the mindbinding of tradition, brought in Western science, the intellectual conquest was swift. Science punctured the fragile structure of ancient traditions, and China joined the twentieth century. Then in 1954 the communists overturned all these advances, exiled the intellectuals to farms and brought back superstition and ignorance as state policy.

China story, version 2.0: China, the oldest civilization on the planet, had a rich structure of tradition. In the twentieth century, overawed by Western technological advances, young Chinese revisionists built a power structure of intellectuals modeled along alien Western lines, and trampled traditional Chinese values. In 1954, the government took action to redress the balance and protect the country's traditions from Western erosion.

Which China story is the truth? Probably neither, but both have been purveyed as truth, and the disparity between the two versions underscores the danger of complacency about one's worldview. Western science is itself a tradition, with its blind spots and unquestioned dogma. It isn't always the best tradition. It is now apparent that some traditional Chinese medical practices work better than their Western counterparts. Unfortunately the blinders of the Western scientific tradition insured that only those scientists who were willing to look into something they *knew* couldn't work were in a position to learn this fact. Those who entertained notions that bordered on fantasy are the ones who learned something new.

It's that way with programming, too.

DDJ isn't interested in *just* helping programmers write better spreadsheets and word processors (although we do that.) Programming at its best is a creative activity, and requires that willingness to entertain ideas that can't possibly work. Programming at its best is what *DDJ* is about. So we'll range beyond the fields we know, and sometimes we may trip. This time, apparently, we didn't. As of this writing, our mail is running overwhelmingly in favor of the sixth generation article.

This month we set forth to China, and embark on a different kind of fantasy, with Timothy Huang's article on Chinese Forth. *DDJ* has always championed the cause of putting computer power into the hands of the largest number of people, and placing Forth in the language of a large fraction of Earth's inhabitants is surely a step in that direction. It is of course a fantasy to think that the efforts described in this article will have any significant effect on U.S.-Chinese relations or on our survival on the planet. Isn't it?

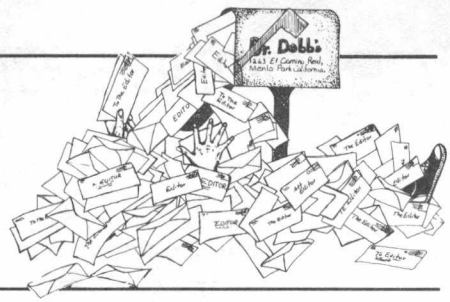
Oh yes, the dragons.

This month's cover depicts two dragons, one Western, one Chinese, sharing technology. (The Western dragon is the fat one.) It nicely illustrates our Chinese Forth article. But the dragon is *DDJ*'s symbol, too; one the magazine has, since its first issue, used as a reminder not to be tied to the fields we know. Personal computers are changing the world, and only those who are willing to poke the dragon will find the jewels clinging to its belly.

DDJ has been tickling the dragon for eight years, and sharing the jewels. We think we'll keep right on doing it.

Michael Swaine

LETTERS



The editorial response card is a great way to talk to us, but don't forget that Dr. Dobb's Journal *also welcomes letters to the editor as a forum for ideas, innovations, irascibility and even idiosyncrasies. Some letters may be edited for clarity and brevity. The Doctor likes hearing from you – keep on writing.*

On Soul of CP/M

Dear *DDJ*:

I get the feeling that the review of *Soul of CP/M,* Waite & Lafore, was from the viewpoint of an experienced assembly language programmer looking backward. Granted, the number of editorial errors *was attackable, but* the manner in which the material was presented was tremendous for the target readership.

I've read (or started to read) a good number of CP/M assembly language books. Nealy all of them take the approach of presenting the BDOS calls at the front, numerous multi-page program samples with unintelligible labels and explanations, some cryptic references to the perfect BIOS, and an appendix with the 8080/Z80 instruction set.

Waite & Lafore feed the material in chewable bytes (pun intended) so the reader doesn't need a brain with a Z80 and 64K of permanent memory to comprehend and retain it. No, it is certainly not a complete text, but at least now all those other books make sense to me.

If they can eliminate the remaining editorial errors in the second printing, they have a winner.

Doug Hurst
6808 Estrella Avenue
29 Palms, CA 92277

Even Better Preprocessing

Dear Doctor:

Although this journal may not have many clients – excuse me, readers – who do (or will admit to doing) much programming in BASIC, there must be others like myself who for one reason or another are constrained to use this language (but only during working hours, I protest). All BASIC users owe a debt of gratitude to N. C. Shammas for the NBASIC preprocessor presented in your January issue. This tremendously useful routine has mitigated some of the most serious drawbacks of interpreted BASIC.

There are, however, a couple of improvements that users might wish to make. The first of these is to make the CASE statement accept strings as expressions, a relatively simple change that requires adding or modifying only four or five lines. The second is a major rearrangement of the program so that CALL statements are processed before the CASE-OF structures. The reason for this is that processing of the CASE-OF commands puts a GOTO on the end of the last line of each group of expressions. CALL statements are resolved into GOSUBs.

Consider, then, the situation when a CALL is the last line of a CASE-OF expression group. As the program is written, the GOSUB is never executed because it becomes preceded by the GOTO. There is another related problem that arises from the appending of a GOTO on the last line of a CASE expression group. If the last line of this group is an IF statement, there must be an ELSE clause included *if* the CASE construction includes an !ELSE DO statement group. Otherwise, when the IF statement is false, the GOTO will not be executed and the !ELSE DO statement group *will* be executed. With these two modifications made, and the last quirk kept in mind, NBASIC can quickly become an indispensable part of programming in this most widespread of languages.

Both N. C. Shammas in particular and *Dr. Dobb's* in general deserve great praise for the quality and usefulness of their articles.

Sincerely,
Dreas Nielsen
234 NW 30th St.
Corvallis, OR 97330

Remarks on RSA

Dear Sirs:

Thank you for a most interesting article on the Rivest-Shamir-Adleman (RSA) Public Key Systems by C. E. Burton in the March 1984 issue. This discourse is well written and certainly very needed. I would, however, like to caution the readers against too heavy a reliance on the security aspects of RSA.

Earlier this year Gustavus Simmons, James Davis, and Diane Holdridge of Sandia National Laboratories were able to factor the last of the Mersenne primes (2**251)-1. By using parallel processing of number clusters it is simple to extrapolate that RSA will no longer be a secure system.

Undoubtedly for general use RSA will continue to be valid. However, with some of these newly developed techniques the only truly secure system must be the old, one-time pad system, in which the volume of ciphertext is insufficient to analyze statistically, and the key changes with a frequency to prevent valid analysis.

A good basic text for your readers so interested is *Cryptography, A Primer* by Alan G. Konheim (John Wiley & Sons, 1981).

Please keep up the good work. I certainly look forward to your publication each month.

Cordially,
James R. Criscione Jr., M.D.
President
Med-Data, 246 Grand Avenue
Kirkwood, Missouri 63122

Dear Dr. Dobb:

It was good to hear about an old friend, RATFOR, in Mr. Burton's article on the PKS in the March '84 issue of *Dr. Dobbs.* I have run several tens of thousands of lines of code through the RATFOR preprocessor and was very grateful for its services in an environment where Fortran was the only compiler available. But I am somewhat ambivalent about promoting its usage when there are viable alternatives.

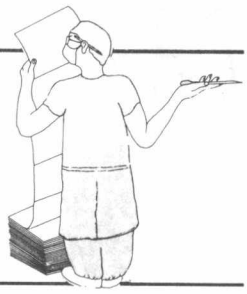
My negative feelings toward RATFOR are summarized by the following factors:

(1) relatively few people use RATFOR (or have even heard about it) and it is difficult to find a community of users for sharing experiences or software,
(2) installation of RATFOR can be difficult and occasionally short circuited by versions of Fortran with nonstandard limitations,
(3) the added time for preprocessing on a small micro can be excessive,
(4) temptations to hack RATFOR into "new and improved" versions can destroy code transportability, and
(5) we are still using Fortran with all its fundamental limitations.

Fortran has far outlived its useful purpose (if it ever had one) but it is, admittedly, one of the most commonly available compiler languages on minis and midis and maxis. I heartily recommend RATFOR as a salve to soften the agony of having to deal with this unwieldy Tyrannosaurus. Mssrs. Kernighan and Plouger managed to make far better use of their

(Continued on page 16)

DR. DOBB'S CLINIC



by D.E. Cortesi, Resident Intern

MiniReview: Quasi-Disk

Late last year we decided to reward our S-100 system for three years of faithful work by giving it some new hardware. One thing it had been pining for was an electronic drive. It would be nice to say that we did a careful engineering analysis of the products on the market, but in fact – tell the truth and shame the devil – we merely checked the prices of all the RAM-drive boards we could find in the ads in one month's worth of computer magazines.

The cheapest one was the Quasi-Disk, from Electra-Logics, Inc. (39 Durward Place, Waterloo, Ontario, Canada N2L 4E5; (519) 884-8200). Its retail price of $799 for 512K was, in December 1983, half that of its nearest competitor. Never ones to let engineering details blind us to the bottom line, we ordered one, plus a battery backup unit at $159.

The order was held up because of a parts shortage, but Electra-Logics was decent enough to hold our check, only cashing it after it shipped the board early in February. The board arrived in mid-February, and the battery backup unit trailed in in early March.

Once it had arrived, the Quasi-Disk gave a good account of itself. Following the manual, we cut one trace (pin 53), which appears as ground in the IEEE standard but as a disable signal in our system. That done, we slotted the board into the system and fired it up, and the system worked normally. The package included a good diagnostic program that ran correctly under CP/M Plus; it reported that the board was working.

The Quasi-Disk package contains a lot of software (including a self-installing disk driver and a print spooler), but all of it is designed to work with CP/M 2.2. We couldn't test it because we are on CP/M Plus exclusively. It took us several hours to ferret out the details of how to read and write the board as a pseudo-disk and to create a new module for the CP/M Plus BIOS. Installing the code wasn't hard, however, since the CP/M Plus BIOS is modular. All the Quasi-Disk code went into a separate assembly; the names of its public entry points went into the drive table; the BIOS was relinked and a new system generated; and we were up.

We made the board appear to the rest of the system as a disk of 64 tracks, each holding eight 1K physical sectors. The distributed software treats the "disk" as one having 128-byte sectors, but we wanted to reduce the traffic between the BDOS and the BIOS. Whereas the distributed code uses a 1-byte checksum for each "sector," we reserved 2K of the board to hold true 16-bit CRC codes. No CRC errors have been trapped in six weeks of operation.

The Quasi-Disk is much faster than a mechanical drive, but access to it isn't instantaneous. Data transfer takes just as long as it does to a real disk, but there are no seek or rotational delays. We have a disgustingly disk-bound Sort program. When it sorts a 60K file on disk, it takes 15 minutes 25 seconds. When its work files are on the Quasi-Disk, it runs in 9 minutes 25 seconds, 61% of the mechanical time. The extra six minutes of the disk sort are entirely devoted to head motion and rotational delays – a sobering thought.

Speed is not the Quasi-Disk's only benefit. It turned out to be very nice just to have a third drive. Under CP/M Plus, the PROFILE.SUB that runs automatically at boot time can load that drive with the 100K of standard utilities, then set the drive-search path so that the system searches for commands first on drive M, then on drive A. That lets us omit the standard commands from other disks but still allows us to access them, quickly and silently, at any time.

The battery backup unit comprises a transformer that plugs into the wall and a black box, the size of a carton of cigarettes, that is stuffed full of sealed gel cells. A ribbon cable from the black box snakes into the computer, where it plugs into the Quasi-Disk. When system power is off, the transformer keeps the board alive. If the electric service fails, the gel cells can take over for a couple of hours. A longer outage drains the batteries, losing data.

In our judgment, the Electra-Logics Quasi-Disk is a sound, well-made piece of hardware at a very good price. Its documentation is complete, although not well organized. Its supporting software is entirely oriented to CP/M 2.2, so the CP/M Plus user has to be able to roll his own. That's a pity, because it's the flexibility of CP/M Plus that makes the Quasi-Disk a really convenient extension of the system.

Fooling RMAC

Relocating assemblers such as Digital Research's RMAC make a distinction between relocatable values and nonrelocatable ones. The difference is hard to grasp at first, and if you fail to grasp it, you may be sorely puzzled by the assembler's actions. Even when you do grasp it, the assembler's rules on what you can and can't do with a relocatable value can cramp your style.

A label is a relocatable value. Define a label:

```
Here:  mvi a,1
```

Now the value of the symbol "Here" has been set as a 16-bit number, the contents of the assember's location counter at that point. The number is a relocatable value, however. The value of "Here" is only the relative offset of "Here" within its particular segment. "Here" won't be at that location when the program runs; its actual storage address will be offset by some amount that can't be known at this time.

Define two labels in the same segment:

```
Here:  mvi a,1
There: ora b
```

Both are relocatable. The assembler will permit you to subtract them,

```
dw There-Here
```

because the difference between them is a constant – it will be the same no matter where the linker puts the program. That is true only when the labels are defined in the same segment, of course. The relative origins of the code and data segments can't be known at assembly time, so the difference between a code label and data label can't be known.

The assembler also can allow you to add or subtract a constant from a relocatable value, so:

```
dw There-16
dw Here+5
```

It can permit that because a linker that supports the standard REL-file format must permit adding a constant to a relocatable value. The algebraic addition will be performed at link time; the assembler only needs to record a reference to the (future) location of the label and to record the constant.

No other manipulations are permitted on relocatable values. The value of, say, Here AND OFFh can't be known at assembly time. Nor can the sum of two relocatable values be known. You can usually get around these things, if only by coding instructions that will carry out the evaluation when the program is executed.

What can't be gotten around is the need to evaluate macro operands that are relocatable values. In particular, RMAC doesn't allow any relocatable value to be

evaluated in an IF statement.

This caused us a problem. We were making a set of macros in which we wanted either to save and load a register or leave it alone before doing something. In other words, we wanted to code something along these lines:

```
if ... some test of an operand ..
    push  d
    lxi   d,operand
endif
```

One way to do this is to allow an omitted operand to mean one thing and a stated operand the other:

```
if not nul operand
    push  d
    lxi   d,operand
endif
```

But omitting operands is tricksy and makes the code hard to read. It would be a lot nicer, and a lot harder to err, if the user had to write some special keyword – say, "noload" – to specify that the register is already loaded.

Now, the DRI assemblers provide no method by which you can test the literal, superficial encoding of a macro operand. There is no way to code the test, "If the user wrote exactly 'noload' then...." Some assemblers for the 8086 do allow that, using the angle brackets as metaquotes. But in MAC and RMAC, if you code

```
if operand ne noload
```

the assembler will compare the assigned values of "operand" and "noload," *not* the character strings that name them.

Very well, we thought, we will define our keywords in equate statements, giving them highly unlikely values, e.g.,

```
noload equ 0abcdh
```

Now we can write

```
if operand ne noload
```

and the assembler will compare the value substituted for "operand" to the value equated to "noload," and we will have our test.

Nope. The value assigned to "noload" is nonrelocatable. If the value substituted for "operand" is relocatable, the IF statement will produce an error message, because we are trying to mix relocatable and nonrelocatable values in one expression.

Oh dear, are we stumped? Not for long. A careful reading of the rules for the substitution of macro operands into macro parameter names reveals an out.

If a macro operand is coded with a leading percent sign, the assembler will evaluate it and substitute not the value with its attached attribute of relocatability, but a simple string of decimal digits. That leads us to the following relocation-stripping macro:

```
unrel macro operand
    @unrel set operand
    endm
```

It does nothing but store the value it was given in a global name. The trick is in how it is used. Within our main macro, where we want to compare for the value of keyword, we now write

```
    unrel %operand
if @unrel ne noload
    push  d
    lxi   d,operand
endif
```

The percent sign strips the attributes from "operand." Whatever happens to be substituted for it, be it a relocatable label or not, the value assigned to "@unrel" is a simple constant. That can be compared to the equated value of "noload" without error. The full treatment ends up as:

```
some macro operand
    if not nul operand
        unrel %operand
        if @unrel ne noload
            push  d
            lxi   d,operand
        endif
    else
        +++ operand required as
            address or 'noload'
        exitm
    endif
    ... etc, etc,
    if @unrel ne noload
        pop d
    endif
endm
```

The user is forced to code some operand. If the operand is omitted, the invalid expression "+++" forces the assembler to display the message line. In that line, the keyword is quoted. If it weren't, the assembler would substitute for it, producing the error message "operand required as address or 43981," 43981 being the value 0abcdh equated to "noload."

DIR Not-Quite-So-Full

CP/M Plus has an extensive DIR command, one that can produce a wide variety of displays. It is very useful, but it is also a trifle verbose. When you ask for a full display, you get:

- a blank line
- a line, "Scanning Directory"
- another blank line
- a line, "Sorting Directory"
- another blank line
- column headings
- another blank line
- a line of dashes
- another blank line

Then, finally, you see the display of files, sizes, timestamps, etc. These extra lines are unnecessary. Often they push off the screen previous commands that one would like to refer back to.

We worked out how to stifle some of these lines. If you (the six of you that have CP/M Plus) would like to stifle them as well, you can do it this way. Load up DIR.COM under SID, using this sequence:

```
rename dir.old=dir.com
sid dir.old
```

Check these five hex addresses for the given instructions:

```
2091  CALL 30E2
24F7  CALL 30E2
2DF0  CALL 272C
2DF6  CALL 272C
2E2D  CALL 272C
```

If your DIR.COM is the same as ours, that's what you'll find. The first two calls produce the "Scanning" and "Sorting" messages and the blank lines that follow them, while the others produce blank lines around the heading.

Replace each CALL with three NOP instructions, then write a new DIR file, using

```
-wdir.com
```

Test the command. It should produce a more compact display.

Without Fear or Good Taste

Dear Reader, the ol' mailbag is running dry. What has your system done to surprise you lately? What has your operating system done to disgust you, and how did you get even with it? Know any good puzzles? Gotta have input, gotta get some of the cobwebs off the chairs in the waiting room at the old Clinic.

Hey, I know, let's have a contest! (Picture Mickey Rooney: "Hey, gang, I know – we'll put on a *show!*") Sure. Let's hold the first (and last) annual ZOSO sound-alike contest. Surely you remember ZOSO, the vile-tempered person who used to hold forth with outrageous, near-libelous, dead-accurate opinions on the old CPMUG disks?

If you can't think of anything else to write about (puzzles, system discoveries, perils and pitfalls), then send us your best imitation of ZOSO: a page of scornful, acerbic, sneering commentary on some aspect of the personal-computer scene. We'll print the best ones that our lawyer will approve. You'll feel *much* better afterward.

■DJ

CP/M on the Commodore 64

Including Two BIOS Modifications

After the recent outcry about the correct spelling of the word kernel, *we thought it interesting that the documentation for the Commodore 64 spelled the word* kernal. *Because this article refers to the Commodore "kernal" specifically, we have decided to use Commodore's spelling in this instance. – Ed.*

The Commodore 64 has the potential to become one of the least expensive "real" computer systems available. Although most people know that the C-64 has a poor version of BASIC and that the OS doesn't really qualify as an operating system, not as many people know that they don't have to be a captive of Commodore's firmware.

The C-64 contains 64K of real RAM. Some of it is hidden under the ROM that holds the BASIC and the ROM that holds the OS kernal, but any program can switch out both of these ROMs to give access to the RAM underneath. Commodore is marketing a CP/M package for the C-64 that takes partial advantage of this hidden RAM by switching out the BASIC but leaving the OS I/O kernal switched in. Their implementation results in a computer that can run a 48K version of CP/M 2.2 (Although it is not the subject of this article, I'm convinced that a larger version of CP/M is possible with a rewrite of the BIOS.)

The CP/M package, which is similar in concept to the CP/M for the Apple, consists of a Z80 co-processor that plugs into the expansion port on the back of the keyboard and a CP/M system diskette. When the C-64 is running CP/M, the Z80 and the C-64's native processor (the 6510) take turns running out of the common memory. The 6510 becomes a slave processor; the Z80 turns it on whenever it requires I/O. Communication between the two processors is through data and commands left in memory. As with all other CP/Ms, only the BIOS is customized for the Commodore; the BDOS is not aware of the existence of the 6510.

by Walt Piotrowski

Walt Piotrowski, R. D. 1 Box 582, Afton, NY 13730.

The CP/M system is started with the C-64 running in its native mode. After plugging the CP/M cartridge into the back of the C-64, the user loads a CP/M boot program, called CPM, from the system diskette just as if it were a BASIC program. This boot program, called BOOT65 in the Commodore CP/M manual, loads in two other routines. The first is the 6510 portion of the BIOS (BIOS65), and the second is a Z80 boot routine (BOOT-80). Once both routines are in place, BOOT65 switches on the Z80, and from this point on the Z80 is in charge. The Z80 boot routine loads in the Z80 portion of the BIOS (BIOS80) and the rest of CP/M.

The interface between the two parts of the BIOS is very nicely designed and very clean. Ten memory locations serve as a communication region, and a 256-byte buffer holds a full C-64 disk sector. When the Z80 part of the BIOS needs some physical I/O, it puts the required command information in the communication area (locations $900-$90A in the 6510 address space), switches itself off, and turns the 6510 on. The 6510 picks up the command, does what is required, then switches back to the Z80. The hardware is arranged so that both computers resume execution at the instruction following the instruction that performed the switch (plus one byte).

Because both processors required access to a set of low-numbered memory addresses to function reasonably, the address spaces in the two machines are offset from each other. The 6510 keeps its low memory data in the location adjacent to the "real" zero memory location. As shown in Figure 1 (page 15), the zero memory location for the Z80 is actually at location $1000 in this 6510 address space. Because of the offset of $1000, the communication area at $900 in the 6510 address space is at location $F900 in the Z80 space (see Figure 2, page 15). In other words, $1000 is added to every Z80 address; hence, a Z80 access to $F900 yields $0900, and a Z80 access to $0000 reaches $1000.

Adding a Second Drive

I get the impression that Commodore rushed to market with CP/M. The software and the documentation both have some shortcomings. The biggest problem is that the BIOS can communicate with only one disk unit. It can talk to both drives in a Commodore 4040 unit or to the single drive in a 1541, but it cannot communicate with two or more 1541 drives. If you have a 1541, you can still use an A and a B "drive" in CP/M, but each time you switch between A and B, you have to swap diskettes in and out of a single 1541. The software in the BIOS always tells you when the swap is required. I have two 1541s, however, and I quickly became annoyed with diskette swapping in one of them while the other sat unused. What I did about it follows.

The manual supplied by Commodore contains complete listings of all of the Z80 code for CP/M, but it does not contain listings for any of the 6510 code. The Z80 listings are well commented and easy to read. Because of this and because of the cleanliness of the interface between the two parts of the BIOS, I was able to follow the 6510 code with the use of a disassembler.

The changes that I made to allow CP/M to use two 1541 drives are relatively simple. They are so simple that I am surprised that Commodore did not include this capability in the original package. These modifications are shown in Listing One (page 18). The first four sections of this listing are patches that replace some of the original BIOS65 code. The last sections are small additional routines that are called by the patches; they fit in a section of memory reserved for the BIOS but actually unused.

The first patch routes the processing of all BIOS65 commands from the Z80 through the new routine called TESTIT. This routine checks to see if the command is a disk operation. If it is, the routine compares the drive number in the command with the drive number that was accessed on the last disk command. If they are the same, control returns to BIOS65. If they are different, the routine just closes the disk channels then returns control to BIOS65. BIOS65 discovers that the channels are closed when it attempts to communicate with the disk and tries to reopen them. The last two patches route the open requests through the new routines called OPEN15 and OPEN2, which open the channels to the correct disk. The changes make minimal impact on the original program. All of the disk errors that are detected and reported back to CP/M still function as they did originally.

A bigger part of the problem was coming up with a way to make the changes permanently on a CP/M diskette.

Since my 6510 assembler and loader do not run under CP/M, I chose to make all of the changes on the 6510 in its native mode. This required writing a utility program to read the 6510 code from the diskette and to replace it, once it had been modified, back in the same sectors. This utility turned out to be longer than the changes themselves, although it was easier to write because I didn't have to explore someone else's machine language code first.

The utility (CPM65UT) is provided in Listing Two (page 19). The utility does not have the capability to make changes; it simply reads the 6510 code from the disk or replaces it. Therefore the utility must be used in conjunction with a loader or a machine language monitor. The 6510 code is read into memory by calling the utility at its read entry point ($C000). It reads BOOT65 and BIOS65 into the locations where they normally reside (see Figures 1 and 2). The loader or monitor is then used to make the modifications. After the changes have been made, the utility is called at its write entry point ($C003), and it writes the modified code back onto the disk. Its use with the loader in the Commodore Assembler Development System is discussed at the end of this article.

Adding a Serial Printer

My printer is not a Commodore printer. It's a DECwriter that I've had for several years; I modified it for RS-232 operation when I bought the C-64. While I was modifying the BIOS for two-disk operation, I also made a change to incorporate the DECwriter as the system printer. Listing Three (page 26) shows the changes to BIOS65 that make this substitution. The two changes, the one for the disks and the one for the RS-232 printer, are completely independent; you can incorporate either of them separately or use both together.

If you examine the Commodore CP/M manual, you will find that Commodore provides two user-definable functions in the interface between the Z80 and the 6510 parts of the BIOS. They also leave two 256-byte areas in the space reserved for BIOS65 to allow room for the 6510 code that you will write for these functions. Because I wanted to use all of the 48K bytes of memory that are available for CP/M programs and because I had no plans to implement any user functions, I moved the Commodore kernal's RS-232 buffers into one of these areas. Their normal location would have decreased the space available to CP/M by 4K bytes. If you choose to make the changes the same way I have, you will not be able to use the second user function in any program that also uses the printer.

Although it is unlikely, the possibility exists that more than one version of Commodore's CP/M is in circulation. Therefore, I've included comments in the two-disk modification to show the original memory contents for each of the four small patches. If you decide to make the changes, be sure to compare what is in your memory with what the listing says should be there. If you find a difference and need some help, please let me know and I'll do all I can to assist you. If you decide to make the RS-232 modification, you should note that it is a large patch that completely overlays the printer code in the BIOS. If you find that the disk patch locations match up, you should be able to make the RS-232

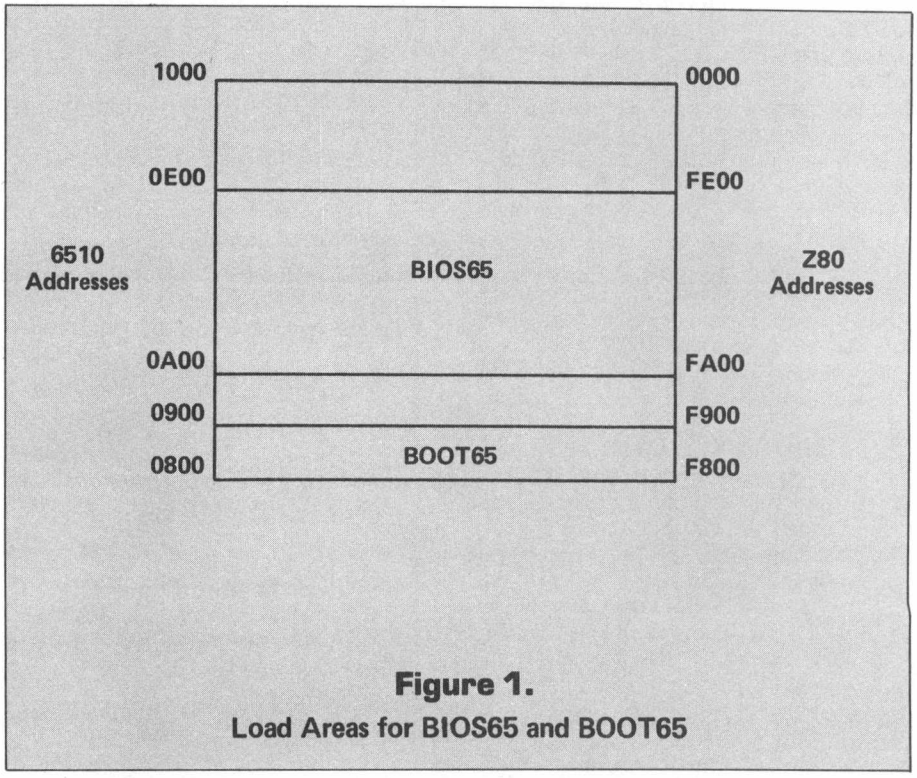


Figure 1.
Load Areas for BIOS65 and BOOT65

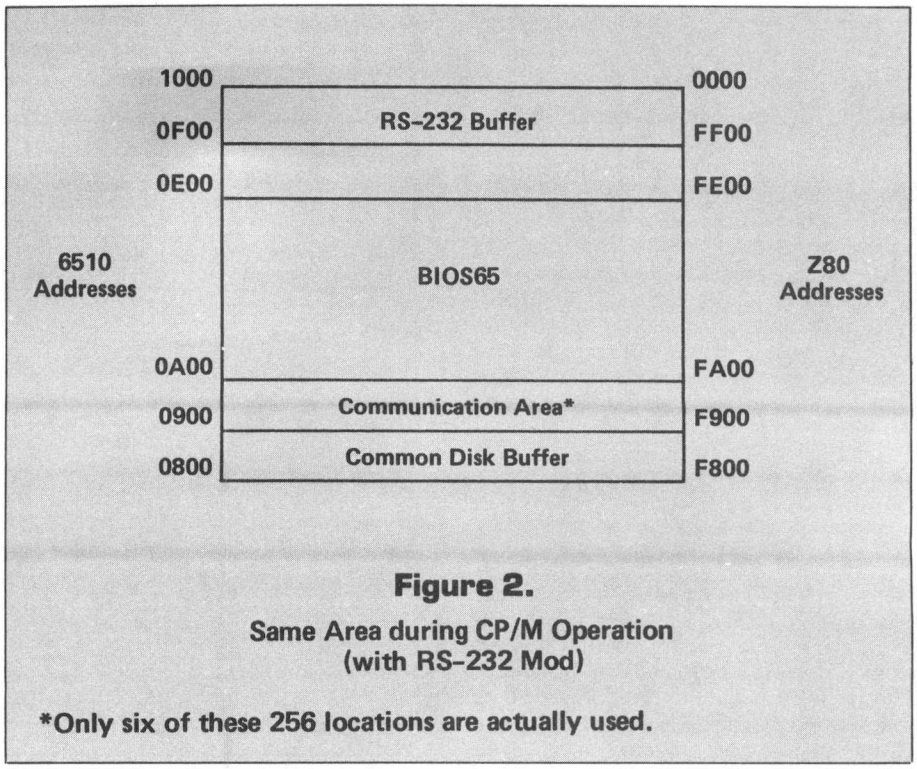


Figure 2.
Same Area during CP/M Operation (with RS-232 Mod)

*Only six of these 256 locations are actually used.

change easily. If you have a problem, again I will be happy to help out.

After you have made the changes to the BIOS, you must run the CONFIG program that is located on the CP/M diskette. If you have made the disk drive changes, you should change to a two-drive configuration. The modifications made to the BIOS will allow you to change back and forth at any time. If you have CP/M configured for two drives, the program will access drives 8 and 9; if you have CP/M configured for one drive, the program will use drive 8 in the diskette swapping mode. (It is also possible to use more than two 1541 drives by making a minor change in BIOS80.) If you have made the RS-232 change, you should use CONFIG to change the CP/M configuration so that BIOS80 thinks that you have the Commodore 4022 printer. If you leave it configured for the 1525 printer, BIOS80 makes a conversion from the ASCII character set that is used by CP/M to the 1525's character set.

You should be cautious about two things if you decide to buy Commodore's CP/M. The first is memory size. If you look through the CP/M Software Finder, you will find many programs that will run in 48K, but many also will not. All of the ones that I am now interested in fit in 48K, but this may not be true for you. The second is disk format. Commodore's 1541 disk format is unique, and the software supplier for the program that you are interested in may not have this format available yet. I believe that this is a temporary problem and that it will be solved as the suppliers begin to realize that there are a lot of C-64 owners in the world.

As a final note, there is also the possibility of downloading public domain software from a bulletin board or another CP/M system using the inexpensive modems available for the C-64. I'm not aware of any software currently available for Commodore's CP/M that will do that, but I would like to hear from you if you know of any. If not, it doesn't look as if it would be that tough a program to write, making use of the user functions that are provided in the interface between BIOS65 and BIOS80. Any volunteers?

Using CPM65UT with the Commodore Assembler Development System

The Commodore Assembler Development System is a software package that contains an editor, an assembler, a machine language monitor, and a loader, along with several other useful programs. The package contains two versions of the loader. The version called HILOADER64 resides in memory at $C800 (51200 decimal). If you have previously assembled the CP/M changes, you can load both the CPM65 utility and the loader into memory at the same time. You then execute the utility by typing SYS49152. The utility will read the 6510 code into memory. When this is complete, you execute the loader by typing SYS51200. The loader will ask the name of the object file that you want to load. When loading is complete, you write the modified code back onto disk by typing SYS49155.

References

Commodore Business Machines, *Commodore 64 CP/M Operating System User's Guide*, Howard W. Sams and Co., Indianapolis, IN 46268, 1983.

Commodore Business Machines, *Commodore 64 Programmer's Reference Guide*, Howard W. Sams and Co., Indianapolis, IN 46268, 1983.

Digital Research, *CP/M Software Finder*, Que Corporation, Indianapolis, IN 46250, 1983.

Rodnay Zaks, *The CP/M Handbook with MP/M*, Sybex Corp., Berkeley, CA 94710, 1980.

DDJ

(Listings begin on page 18)

(Continued from page 10)

time developing RATFOR than the committee that came up with Fortran '77.

An initial perusal of the article also revealed a typo in the multiplication routine descriptive text: the line "M5" i index should read "i=i-1" not "i=i-i". I also would suggest that Mr. Burton can save some computer time by removing the "mod" and division operations from the multiple-precision add and subtract functions. By using the facts that the sum is always going to be less than twice the radix and the absolute value of the difference is always less than the radix, the C code in Figure (at right) shows "division/mod-less" evaluations.

If we declare:

```
long carry, acc;
unsigned short *u, *v, *w;
```

then we can define RADIX to be "power (2, n)" (2**n for Fortran folks) where n is the number of bits in a type "short" word. This will allow us to compute the equivalent of about 4.5 decimal digits per loop iteration on a machine with 16-bit type "short." Of course, when debugging the system we can define RADIX as 10 to facilitate formatting. I would also tend to make RADIX a compile-time definition rather than a runtime argument for the PKS software.

The last comment relates to Mr. Burton's statement at the end of the addition algorithm text that the Fortran intrinsic "INT(x)" is equivalent to "floor (x)." This is only true for positive values of x. "floor" (related to ALGOL's "entier") is correctly stated as the greatest integer less than or equal to x. Consequently:

floor(-3.45) -> -4

But the Fortran intrinsic yields:

INT(-3.45) -> -3!

I didn't see that Mr. Burton ever used it, but the statement was rather careless and should have been caught in technical review.

Obviously I've found Mr. Burton's article interesting and intend to follow along the steps to producing a public key encryption system as a diversion from some of my more mundane programming activities. It takes me back to the "secret" messages of Capt. Midnight, *et al., and* the kids' radio shows of the '40s. The trouble is, our decoding rings have gotten a little more expensive. Maybe Dr. Dobb can have "secret" messages about next month's articles.

Sincerely,
Gerald I. Evenden
Box 1027
North Falmouth, MA 02556

DDJ

```
carry = 0;
while (w >= ws) { /* basic addition loop */
     acc = *u-- + (*v-- + carry);
     *w-- = (carry = acc >= RADIX) ? acc - RADIX : acc;
}

carry = 0;
while (w >= ws) { /* basic subtraction loop */
     acc = *u-- - (*v-- - carry);
     *w-- = (carry = acc < 0) ? acc + RADIX : acc;
}
```

Figure

CP/M on the Commodore 64 (Text begins on page 14)

Listing One

```
CPMDSK.TX.

LINE# LOC   CODE        LINE

00001  0000             ;************************************************
00002  0000             ;
00003  0000             ;     MODIFICATIONS TO BIOS65 FOR
00004  0000             ;     TWO 1541 DISKS (8 AND 9)
00005  0000             ;
00006  0000             ;     W.G. PIOTROWSKI
00007  0000             ;     STATE UNIV OF NY
00008  0000             ;     BINGHAMTON, NY 13901
00009  0000             ;
00010  0000             ;************************************************
00011  0000             ;
00012  0000             BIOS65 =$0A0C          ;BIOS RETURN LOC
00013  0000             COMAND =$0900          ;COMMAND LOC
00014  0000             DISKNO =$0904          ;DISK NUM LOC
00015  0000             CLOSE  =$FFC3          ;KERNAL CLOSE ROUTINE
00016  0000             ;
00017  0000                    *=$0A06         ;CHECK FOR DISK OPS
00018  0A06  20 B0 0C         JSR TESTIT      ;WAS  JSR $0A0C
00019  0A09             ;
00020  0A09                    *=$0AFB         ;REMOVE DISK NUM STORE
00021  0AFB  EA               NOP             ;WAS  STA $0B67
00022  0AFC  EA               NOP             ;      "    "
00023  0AFD  EA               NOP             ;      "    "
00024  0AFE             ;
00025  0AFE                    *=$0B9C         ;BIOS TRYING TO OPEN 15
00026  0B9C  20 90 0C         JSR OPEN15      ;WAS  LDA #15
00027  0B9F  EA               NOP             ;     LDX #8
00028  0BA0  EA               NOP             ;     LDY #15
00029  0BA1  EA               NOP
00030  0BA2             ;
00031  0BA2                    *=$0BB6         ;BIOS TRYING TO OPEN 2
00032  0BB6  20 A0 0C         JSR OPEN2       ;WAS  LDA #2
00033  0BB9  EA               NOP             ;     LDX #8
00034  0BBA  EA               NOP             ;     LDY #2
00035  0BBB  EA               NOP
00036  0BBC             ;
00037  0BBC                    *=$C90          ;FREE SPACE IN BIOS AREA
00038  0C90  A9 0F      OPEN15 LDA #15         ;COMMAND CHANNEL
00039  0C92  20 C3 FF         JSR CLOSE       ;CLOSE-JUST IN CASE
00040  0C95  AD 04 09         LDA DISKNO      ;GET DISK NUM
00041  0C98  18               CLC             ;SET UP FOR ADD
00042  0C99  69 08            ADC #8          ;MAKE 8 OR 9
00043  0C9B  AA               TAX             ;DEVICE # IN X
00044  0C9C  A9 0F            LDA #15         ;CHANNEL IN A
00045  0C9E  A8               TAY             ;ALSO IN Y
00046  0C9F  60               RTS             ;GO BACK-CALL SETLFS
00048  0CA0  A9 02      OPEN2  LDA #2          ;DATA CHANNEL
00049  0CA2  20 C3 FF         JSR CLOSE       ;CLOSE-JUST IN CASE
00050  0CA5  AD 04 09         LDA DISKNO      ;GET DISK NUM
00051  0CA8  18               CLC             ;SET UP FOR ADD
00052  0CA9  69 08            ADC #8          ;MAKE 8 OR 9
00053  0CAB  AA               TAX             ;DEVICE # IN X
00054  0CAC  A9 02            LDA #2          ;CHANNEL IN A
00055  0CAE  A8               TAY             ;ALSO IN Y
```

(Continued on next page)

CP/M on the Commodore 64 (Listing Continued, text begins on page 14)

Listing One

```
00056  0CAF  60              RTS             ;GO BACK-CALL SETLFS
00057  0CB0                 ;
00058  0CB0                 ;
00059  0CB0  AD 00 09     TESTIT LDA COMAND    ;SEE IF DISK COMND
00060  0CB3  F0 08            BEQ CLOSIT      ;0 IS DISK CMD
00061  0CB5  C9 01            CMP #1
00062  0CB7  F0 04            BEQ CLOSIT      ;1 IS DISK CMD
00063  0CB9  C9 06            CMP #6          ;6 IS DISK CMD
00064  0CBB  D0 15            BNE LEAVE       ;NOT FOR DISK-EXIT
00065  0CBD  AD 04 09     CLOSIT LDA DISKNO    ;GET CMD DISK#
00066  0CC0  CD D5 0C          CMP LSTDSK      ;COMPARE WITH LAST USE
00067  0CC3  F0 0D            BEQ LEAVE       ;SAME-LET GO
00068  0CC5  8D D5 0C          STA LSTDSK      ;DIFF-SAVE FOR NEXT
00069  0CC8  A9 0F            LDA #15         ;COMMAND CHANNEL
00070  0CCA  20 C3 FF          JSR CLOSE       ;CLOSE IT
00071  0CCD  A9 02            LDA #2          ;DATA CHANNEL
00072  0CCF  20 C3 FF          JSR CLOSE       ;CLOSE IT
00073  0CD2  4C 0C 0A     LEAVE  JMP BIOS65    ;BIOS WILL OPEN AGAIN
00074  0CD5                 ;
00075  0CD5  00           LSTDSK .BYTE 0
00076  0CD6                    .END
ERRORS = 00000


SYMBOL TABLE

SYMBOL VALUE
 BIOS65   0A0C    CLOSE    FFC3    CLOSIT   0CBD    COMAND   0900
 DISKNO   0904    LEAVE    0CD2    LSTDSK   0CD5    OPEN15   0C90
 OPEN2    0CA0    TESTIT   0CB0

END OF ASSEMBLY
```

End Listing One

Listing Two

```
CPM65UT.TX.

LINE# LOC   CODE        LINE
00001  0000             ;************************************************
00002  0000             ;
00003  0000             ;     UTILITY PROGRAM TO READ OR WRITE
00004  0000             ;     THE 6510 PORTION OF THE CPM BIOS
00005  0000             ;     AND THE 6510 CPM BOOT ROUTINE
00006  0000             ;
00007  0000             ;     W. PIOTROWSKI
00008  0000             ;
00009  0000             ;     TO READ  - JSR $C000   SYS(49152)
00010  0000             ;     TO WRITE - JSR $C003   SYS(49155)
00011  0000             ;       DISK DRIVE # AT $C006   (49158)
00012  0000             ;
00013  0000             ;************************************************
00014  0000             ;
00015  0000             ;     EQUATES
00016  0000             ;
00017  0000             CMDCHN =15           ;DISK COMMAND CHANNEL
00018  0000             DATCHN =2            ;DISK DATA CHANNEL
```

Listing Two

```
00019  0000                ZERO    =$30            ;ASCII ZERO
00020  0000                ONE     =$31            ;ASCII ONE
00021  0000                TWO     =$32            ;ASCII TWO
00022  0000                CR      =$D             ;ASCII RETURN
00023  0000                ;
00024  0000                SETLFS  =$FFBA          ;KERNAL ROUTINES
00025  0000                SETNAM  =$FFBD          ;   "       "
00026  0000                OPEN    =$FFC0          ;   "       "
00027  0000                CLOSE   =$FFC3          ;   "       "
00028  0000                CHKIN   =$FFC6          ;   "       "
00029  0000                CHKOUT  =$FFC9          ;   "       "
00030  0000                CLRCHN  =$FFCC          ;   "       "
00031  0000                CHRIN   =$FFCF          ;   "       "
00032  0000                CHROUT  =$FFD2          ;   "       "
00033  0000                ;
00034  0000                FREKZ1  =$FB            ;FREE SPACE PAGE ZERO
00035  0000                FREKZ2  =FREKZ1+2       ;  4 LOCS AVAIL
00036  0000                BOOT65  =$801           ;NORMAL BOOT65 START
00037  0000                BUF     =$900           ;INPUT BUFFER START
00038  0000                BTSTRT  =$904           ;BOOT65 START IN BUF
00039  0000                BIOS65  =$A00           ;BIOS65 LOAD ADDRESS
00040  0000                BOOTLN  =BIOS65-BTSTRT  ;LENGTH OF BOOT65
00041  0000                ;
00042  0000                ;     MACRO DEF
00043  0000                ;
00044  0000                        .MAC XAD        ;XFER ADDRESS
00045  0000                        LDX #<?1        ;LO ORDER ADDRESS
00046  0000                        LDY #>?1        ;HI ORDER ADDRESS
00047  0000                        STX ?2          ;SAVE LO
00048  0000                        STY ?2+1        ;SAVE HI
00049  0000                        .MND
00051  0000                        *=$C000
00052  C000                ;
00053  C000                ;    ENTRY POINTS
00054  C000                ;
00055  C000  4C 07 C0              JMP READ        ;(49152)
00056  C003  4C 0F C0              JMP WRITE       ;(49155)
00057  C006  09            DISK    .BYTE 9         ;DISK DRIVE #
00058  C007                ;
00059  C007  A9 00         READ    LDA #0          ;READ - ZERO
00060  C009  8D 57 C1              STA INOUT       ;SET FLAG
00061  C00C  4C 14 C0              JMP START       ;GO TO MAIN PROG
00062  C00F  A9 01         WRITE   LDA #1          ;WRITE - ONE
00063  C011  8D 57 C1              STA INOUT       ;SET FLAG
00064  C014                ;
00065  C014                ;       OPEN DISK CHANNELS
00066  C014                ;
00067  C014  78            START   SEI             ;SHUT OFF INTERRUPTS
00068  C015  A9 0F                 LDA #CMDCHN     ;COMMAND CHANNEL
00069  C017  AE 06 C0              LDX DISK        ;DISK ADDRESS
00070  C01A  A8                    TAY
00071  C01B  20 BA FF              JSR SETLFS      ;KERNAL
00072  C01E  A9 00                 LDA #0          ;NO NAME
00073  C020  20 BD FF              JSR SETNAM      ;KERNAL
00074  C023  20 C0 FF              JSR OPEN        ;KERNAL
00075  C026                ;
00076  C026  A9 02                 LDA #DATCHN     ;DATA CHANNEL
```

(Continued on page 22)

Listing Two

```
00077  C028  AE 06 C0          LDX DISK          ;DISK ADDRESS
00078  C02B  A8                TAY
00079  C02C  20 BA FF          JSR SETLFS        ;KERNAL
00080  C02F  A9 01             LDA #1            ;ONE CHARACTER NAME
00081  C031  A2 56             LDX #<FILNAM      ;LO ADDRESS
00082  C033  A0 C1             LDY #>FILNAM      ;HI ADDRESS
00083  C035  20 BD FF          JSR SETNAM        ;KERNAL
00084  C038  20 C0 FF          JSR OPEN          ;KERNAL
00085  C03B                ;
00086  C03B                ;    SETUP FOR READ/WRITE LOOPS
00087  C03B                ;
00088  C03B  A9 30             LDA #ZERO         ;FIRST SECTOR NUMBER
00089  C03D  8D 4A C1          STA USRMSG+9      ;RESET CMD MSG
00090  C040  A9 05             LDA #5            ;FIVE SECTORS
00091  C042  8D 54 C1          STA LOOPCT        ; INITIALIZE LOOP CTR
00092  C045  AD 57 C1          LDA INOUT         ;IN OR OUT?
00093  C048  F0 06             BEQ RD            ;IN - GO TO READ
00094  C04A  20 B0 C0          JSR WRITIT        ;CALL WRITE SUBPROG
00095  C04D  4C 53 C0          JMP EXIT
00096  C050  20 5F C0    RD    JSR READIT        ;CALL READ SUBPROG
00097  C053                ;
00098  C053                ;    WRAP UP AND RETURN
00099  C053                ;
00100  C053  A9 02        EXIT  LDA #DATCHN       ;DATA CHANNEL
00101  C055  20 C3 FF          JSR CLOSE         ;KERNAL
00102  C058  A9 0F             LDA #CMDCHN       ;COMMAND CHANNEL
00103  C05A  20 C3 FF          JSR CLOSE         ;KERNAL
00104  C05D  58                CLI               ;INTS BACK ON
00105  C05E  60                RTS               ;MAIN PROG EXIT

00107  C05F                ;     *** READ FROM DISK ***
00108  C05F                ;
00109  C05F                ;
00110  C05F                ;    READ LOOP - READ 5 SECTORS
00111  C05F                ;
00112  C05F  A9 31        READIT LDA #ONE         ;U1 IS AN INPUT CMD
00113  C061  8D 42 C1          STA USRMSG+1      ;PUT IN THE 1
00114  C064                ;
00115  C064                    XAD BUF,FREKZ1    ;BUF ADDR FOR INLP
00121  C06C                READ1 XAD BPMSG,FREKZ2 ;DISK PTR RESET CMD
00127  C074  A9 08             LDA #BPML         ;MSG LENGTH
00128  C076  20 01 C1          JSR CMDOUT        ;SEND IT
00129  C079                    XAD USRMSG,FREKZ2 ;DISK READ CMD
00135  C081  A9 0B             LDA #USRML        ;MSG LENGTH
00136  C083  20 01 C1          JSR CMDOUT        ;SEND IT
00137  C086                ;
00138  C086  20 1B C1          JSR INLP          ;BRING IN THE DATA
00139  C089                ;
00140  C089  EE 4A C1          INC USRMSG+9      ;POINT TO NEXT SECTOR
00141  C08C  E6 FC             INC FREKZ1+1      ;POINT TO NEXT PAGE
00142  C08E  CE 54 C1          DEC LOOPCT        ;COUNT DOWN TO ZERO
00143  C091  D0 D9             BNE READ1         ;NOT DONE
00144  C093                ;
00145  C093                ;    MOVE "CPM" TO BASIC TEXT AREA
00146  C093                ;
00147  C093                    XAD BTSTRT,FREKZ1 ;SETUP TO MOVE BOOT65
00153  C09B                    XAD BOOT65,FREKZ2 ; TO NORMAL AREA
00159  C0A3  A0 00             LDY #0            ;INITIALIZE LOOPCTR
00160  C0A5  B1 FB        INMOV LDA (FREKZ1),Y    ;GET BYTE
```

```
00161  C0A7  91 FD              STA (FREKZ2),Y  ;MOVE TO WORK AREA
00162  C0A9  C8                 INY             ;LOOP CTR UP
00163  C0AA  98                 TYA             ;INTO A FOR COMPARE
00164  C0AB  C9 FC              CMP #BOOTLN     ;MOVED ALL?
00165  C0AD  D0 F6              BNE INMOV       ;NO-DO NEXT
00166  C0AF  60                 RTS             ;ALL DONE - LEAVE
00168  C0B0                ;    *** WRITE TO DISK ***
00169  C0B0                ;
00170  C0B0                ;
00171  C0B0                ;    LOOP TO MOVE "CPM" BACK TO BUF
00172  C0B0                ;
00173  C0B0                WRITIT XAD BOOT65,FREKZ1 ;SETUP TO MOVE BOOT
00179  C0B8                       XAD BTSTRT,FREKZ2 ;  BACK TO BUFFER
00185  C0C0  A0 00              LDY #0          ;INIT LOOP CTR
00186  C0C2  B1 FB           OUTMOV LDA (FREKZ1),Y  ;GET BYTE
00187  C0C4  91 FD              STA (FREKZ2),Y  ;PUT IN OUTBUF
00188  C0C6  C8                 INY             ;LOOP CTR UP
00189  C0C7  98                 TYA             ;INTO A FOR COMPARE
00190  C0C8  C9 FC              CMP #BOOTLN     ;ALL DONE?
00191  C0CA  D0 F6              BNE OUTMOV      ;NO-DO NEXT
00192  C0CC                ;
00193  C0CC                ;    WRITE LOOP - WRITE 5 SECTORS
00194  C0CC                ;
00195  C0CC  A9 32              LDA #TWO        ;U2 IS AN OUTPUT CMD
00196  C0CE  8D 42 C1           STA USRMSG+1    ;PUT IN THE 2
00197  C0D1                ;
00198  C0D1                       XAD BUF,FREKZ1  ;BUF ADDR FOR OUTLP
00204  C0D9                WRIT1  XAD BPMSG,FREKZ2 ;DISK PTR RESET CMD
00210  C0E1  A9 08              LDA #BPML       ;MSG LENGTH
00211  C0E3  20 01 C1           JSR CMDOUT      ;SEND IT
00212  C0E6                ;
00213  C0E6  20 2E C1           JSR OUTLP       ;OUTPUT THE DATA
00214  C0E9                ;
00215  C0E9                       XAD USRMSG,FREKZ2 ;DISK WRITE CMD
00221  C0F1  A9 0B              LDA #USRML      ;MSG LENGTH
00222  C0F3  20 01 C1           JSR CMDOUT      ;SEND IT
00223  C0F6                ;
00224  C0F6  EE 4A C1           INC USRMSG+9    ;POINT TO NEXT SECTOR
00225  C0F9  E6 FC              INC FREKZ1+1    ;POINT TO NEXT PAGE
00226  C0FB  CE 54 C1           DEC LOOPCT      ;COUNT DOWN
00227  C0FE  D0 D9              BNE WRIT1       ;NOT DONE YET
00228  C100  60                 RTS             ;ALL DONE - LEAVE
00230  C101                ;
00231  C101                ;     SUBROUTINE TO OUTPUT CMD MSG
00232  C101                ;
00233  C101  8D 55 C1     CMDOUT STA MSGL        ;SAVE LENGTH
00234  C104  A2 0F              LDX #CMDCHN     ;COMMAND CHANNEL
00235  C106  20 C9 FF           JSR CHKOUT      ;OPEN FOR OUTPUT
00236  C109  A0 00              LDY #0          ;POINTER TO CHAR
00237  C10B  AE 55 C1           LDX MSGL        ;NUM OF CHARS TO SEND
00238  C10E  B1 FD        CMDLP  LDA (FREKZ2),Y  ;GET A CHARACTER
00239  C110  20 D2 FF           JSR CHROUT      ;SEND IT
00240  C113  C8                 INY             ;NEXT CHAR
00241  C114  CA                 DEX             ;LOOP CTR DOWN
00242  C115  D0 F7              BNE CMDLP       ;NOT DONE - NEXT
00243  C117  20 CC FF           JSR CLRCHN      ;KERNAL
00244  C11A  60                 RTS
00245  C11B                ;
00246  C11B                ;      SUBROUTINE TO READ 256 BYTES
00247  C11B                ;
00248  C11B  A2 02        INLP   LDX #DATCHN     ;DATA CHANNEL #
00249  C11D  20 C6 FF           JSR CHKIN       ;OPEN FOR INPUT
00250  C120  A0 00              LDY #0          ;BUFFER INDEX & LOOP CT
00251  C122  20 CF FF     INLP1  JSR CHRIN       ;GET A CHAR
```

(Continued on next page)

Listing Two

```
00252  C125  91 FB              STA (FREKZ1),Y  ;PUT IN BUFFER
00253  C127  C8                 INY             ;NEXT SLOT
00254  C128  D0 F8              BNE INLP1       ;256 CHARS IN A SECTOR
00255  C12A  20 CC FF           JSR CLRCHN      ;KERNAL CLEAR CHANNEL
00256  C12D  60                 RTS
00257  C12E                ;
00258  C12E                ;     SUBROUTINE TO WRITE 256 BYTES
00259  C12E                ;
00260  C12E  A2 02         OUTLP  LDX #DATCHN    ;DATA CHANNEL #
00261  C130  20 C9 FF           JSR CHKOUT      ;OPEN FOR OUTPUT
00262  C133  A0 00              LDY #0          ;BUFFER INDEX & LOOP CT
00263  C135  B1 FB         OUTLP1 LDA (FREKZ1),Y ;GET A CHAR
00264  C137  20 D2 FF           JSR CHROUT      ;SEND IT
00265  C13A  C8                 INY             ;NEXT CHAR
00266  C13B  D0 F8              BNE OUTLP1      ;256 CHARS IN A SECTOR
00267  C13D  20 CC FF           JSR CLRCHN      ;KERNAL CLEAR CHANNEL
00268  C140  60                 RTS
00270  C141                ;
00271  C141                ;     DATA
00272  C141                ;
00273  C141  55 31 3A      USRMSG .BYTE 'U1:'    ;U1:2 0 1 1
00274  C144  32                 .BYTE ZERO+DATCHN
00275  C145  20 30              .BYTE ' 0 1 1'
00276  C14B  0D                 .BYTE CR
00277  C14C                USRML  =*-USRMSG
00278  C14C  42 2D         BPMSG  .BYTE 'B-P:'    ;B-P:2 0
00279  C150  32                 .BYTE ZERO+DATCHN
00280  C151  20 30              .BYTE ' 0'
00281  C153  0D                 .BYTE CR
00282  C154                BPML   =*-BPMSG
00283  C154  00            LOOPCT .BYTE 0
00284  C155  00            MSGL   .BYTE 0
00285  C156  23            FILNAM .BYTE '#'
00286  C157  00            INOUT  .BYTE 0          ;IN=0, OUT=1
00287  C158                ;
00288  C158                       .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 BIOS65   0A00    BOOT65   0801    BOOTLN   00FC    BPML     0008
 BPMSG    C14C    BTSTRT   0904    BUF      0900    CHKIN    FFC6
 CHKOUT   FFC9    CHRIN    FFCF    CHROUT   FFD2    CLOSE    FFC3
 CLRCHN   FFCC    CMDCHN   000F    CMDLP    C10E    CMDOUT   C101
 CR       000D    DATCHN   0002    DISK     C006    EXIT     C053
 FILNAM   C156    FREKZ1   00FB    FREKZ2   00FD    INLP     C11B
 INLP1    C122    INMOV    C0A5    INOUT    C157    LOOPCT   C154
 MSGL     C155    ONE      0031    OPEN     FFC0    OUTLP    C12E
 OUTLP1   C135    OUTMOV   C0C2    RD       C050    READ     C007
 READ1    C06C    READIT   C05F    SETLFS   FFBA    SETNAM   FFBD
 START    C014    TWO      0032    USRML    000B    USRMSG   C141
 WRIT1    C0D9    WRITE    C00F    WRITIT   C0B0    XAD      FFFF
 ZERO     0030

END OF ASSEMBLY
```

End Listing Two

Listing Three

```
CPM232.TX.
LINE# LOC   CODE        LINE

00001  0000             ;*********************************************
00002  0000             ;
00003  0000             ;       MODIFICATIONS TO BIOS65
00004  0000             ;       FOR AN RS-232 PRINTER
00005  0000             ;
00006  0000             ;       W.G. PIOTROWSKI
00007  0000             ;
00008  0000             ;*********************************************
00009  0000             ;
00010  0000             ;       EQUATES
00011  0000             ;
00012  0000             FILE    =128            ;FILE NUMBER
00013  0000             BUF232  =$F00           ;NEW RS232 BUF LOC
00014  0000             CHAR    =$901           ;CHAR BEING PRINTED
00015  0000             ;
00016  0000             SETLFS  =$FFBA          ;KERNAL ROUTINES
00017  0000             SETNAM  =$FFBD          ;   "       "
00018  0000             OPEN    =$FFC0          ;   "       "
00019  0000             CLOSE   =$FFC3          ;   "       "
00020  0000             CHKOUT  =$FFC9          ;   "       "
00021  0000             CLRCHN  =$FFCC          ;   "       "
00022  0000             CHROUT  =$FFD2          ;   "       "
00023  0000             ;
00024  0000             ENABL   =$2A1           ;RS232 STILL ACTIVE
00025  0000             ROBUF   =$F9            ;RS232 OUTBUF PTR
00026  0000             RIBUF   =$F7            ;RS232 INBUF PTR
00027  0000             ;
00028  0000                     *=$A9F
00029  0A9F             ;
00030  0A9F  AE E2 0A           LDX ENTFLG      ;SEE IF FIRST ENTRY
00031  0AA2  D0 26              BNE OUTCHR      ;ZERO - OPEN FILE
00032  0AA4             ;
00033  0AA4             ;  INITIAL ENTRY
00034  0AA4             ;    OPEN FILE AND MOVE BUFFER
00035  0AA4             ;
00036  0AA4  A2 01      OPENIT LDX #1           ;NOT ZERO MEANS ENTERED
00037  0AA6  8E E2 0A           STX ENTFLG      ;PUT IN FLAG
00038  0AA9  A9 80              LDA #FILE       ;FILE NUMBER
00039  0AAB  A2 02              LDX #2          ;RS-232 DEVICE
00040  0AAD  A0 FF              LDY #$FF        ;NO COMMAND
00041  0AAF  20 BA FF           JSR SETLFS      ;CALL KERNAL
00042  0AB2  A9 02              LDA #2          ;TWO CHAR NAME
00043  0AB4  A2 E3              LDX #<SET232    ;LO ADDRESS
00044  0AB6  A0 0A              LDY #>SET232    ;HI ADDRESS
00045  0AB8  20 BD FF           JSR SETNAM      ;KERNAL
00046  0ABB  20 C0 FF           JSR OPEN        ;KERNAL
00047  0ABE  A2 00              LDX #<BUF232    ;BUFFER LO ADDR
00048  0AC0  A0 0F              LDY #>BUF232    ;BUFFER HI ADDR
00049  0AC2  86 F9              STX ROBUF       ;KERNAL OUT BUF ADDR
00050  0AC4  84 FA              STY ROBUF+1     ;HI ORDER PART
00051  0AC6  86 F7              STX RIBUF       ;MOVE INBUF TOO
00052  0AC8  84 F8              STY RIBUF+1     ; JUST IN CASE
00054  0ACA             ;
00055  0ACA             ;   CHARACTER OUTPUT
```

(Continued on next page)

Listing Three

```
00056  0ACA                    ;
00057  0ACA  A2 80        OUTCHR LDX #FILE        ;GET FILE NUM
00058  0ACC  20 C9 FF            JSR CHKOUT       ;OPEN FOR OUTPUT
00059  0ACF  B0 D3              BCS OPENIT         ;ERROR MEANS CLOSED
00060  0AD1  AD 01 09            LDA CHAR         ;GET CHARACTER AGAIN
00061  0AD4  20 D2 FF            JSR CHROUT       ;KERNAL
00062  0AD7  AD A1 02     WAIT   LDA ENABL        ;GET STATUS
00063  0ADA  29 01               AND #1           ;STILL RUNNING BIT
00064  0ADC  D0 F9               BNE WAIT         ;HANG UNTIL DONE
00065  0ADE  20 CC FF            JSR CLRCHN       ;CLEAR CHANNEL
00066  0AE1  60                  RTS
00067  0AE2                    ;
00068  0AE2  00           ENTFLG .BYTE 0          ;FIRST ENTRY FLAG
00069  0AE3  86           SET232 .BYTE $86,0      ;RS232 PARAMS
00069  0AE4  00
00070  0AE5                    ;
00071  0AE5                    .END

ERRORS = 00000


SYMBOL TABLE

SYMBOL VALUE
 BUF232   0F00     CHAR     0901     CHKOUT   FFC9     CHROUT   FFD2
 CLOSE    FFC3     CLRCHN   FFCC     ENABL    02A1     ENTFLG   0AE2
 FILE     0080     OPEN     FFC0     OPENIT   0AA4     OUTCHR   0ACA
 RIBUF    00F7     ROBUF    00F9     SET232   0AE3     SETLFS   FFBA
 SETNAM   FFBD     WAIT     0AD7

END OF ASSEMBLY
```

End Listing Three

dBASE II Programming Techniques

by Gene Head

Gene Head, 2860 N.W. Skyline Dr., Corvallis, OR 97330.

It seems that everyone nowadays is using a mailing list! Lawyers track clients, clubs chart members, and businesses of all types garner lists of names for mass mailings. I handle a small 1200-name mailing list for a missionary friend who sends field reports to his supporters every month.

Usually, a mailing list will eventually involve the Post Office, and if the list is long enough (200 pieces or more) bulk mailing can save money over regular first class mail. Ah, but there is a catch. The Postmaster expects you to pre-sort your mailing, count it, and have accurate zip code information. Bulk mail is not generally forwarded when an incorrect zip code sends your letter to the wrong city even if the proper city and state are part of the address.

Exact zip code validation is impractical. However, there is a way to partially verify zip codes and validate the state. This technique should decrease errors and increase mass mailing efficiency and accuracy. This procedure could be written in high-level language, but for speed and compactness of code I chose to write it as a machine language subroutine.

ZIP-CHK.ASM assembles into a machine code subroutine that uses a table published by the Post Office to first verify that the two-character state abbreviation is correct then check that the zip code is within the range for that state as set by the table. The source listing is well commented and documented. It was written as a dBASE II subroutine but can easily be adapted to any high-level language that can pass a string type memory variable to the called subroutine and modify that variable if required and pass it back to the calling program.

I got the idea for this program from Bertel Schmitt who wrote an earlier version as part of a file called DBASM.ASM, found on many RCP/M systems around the country. I modified and streamlined the routine, correcting a few errors in the original code. I added a symmetrical look-up table that increased accuracy and speeded the table search.

If you have a modem, you can find ZIP-CHK.ASM on several RCP/M systems, including the dBASE II system at (408) 378-8733. **DDJ**

dBASE II Techniques Listing

```
;          ************************************************
;                    ***** ZIP-CHK.ASM ******
;          ************************************************
;
;    This machine-language subroutine is designed to be called by
;    a high-level language such as dBASE II or BASIC with a single
;    passed variable of exactly five characters like 'ST123', where
;    ST is the Post Office designated two letter abbreviation for
;    the state and 123 are the first three ZIP code digits of that state.
;
;    On return from this sub-routine, the passed variable is un-changed
;    if the ZIP code is valid for the state.  If the two letter
;    abbreviation is invalid then the returned variable becomes
;    'ERR 1' or if the state is valid but the ZIP code is out of range
;    then the variable becomes 'ERR 2'.
;
;    The following command program could be used to verify a state
;    and ZIP code in a dBASE II data file:
;
;         *        ZIP-CHK.CMD
;         *
;         *  Note:    The following two lines of commands should be
;         *           a part of the main program initialization
;         *
;         *        LOAD ZIP-CHK
;         *        SET CALL TO 42096
;         *
;         STORE $(STATE,1,2)+$(ZIP,1,3) TO ZIP:CHK
;         CALL ZIP:CHK
```

(Continued on next page)

dBASE II Techniques Listing (Listing Continued, text begins on page 28)

```
;
;       IF 'ERR'$ZIP:CHK
;       DO ERROR:HANDELING
;       ENDIF
;
;       RETURN
;
;
;
; Adapted from DBASM.ASM file by Bertel Schmitt  (c) 1983
;
; Modified and expanded and accuracy enhanced by:
;
;                                       Gene Head
;                                       Head Quarters
;                                       2860 NW Skyline Drive
;                                       Corvallis, OR  97330
;                                       (503) 758-0279
;
; Last up-date 02-02-84
;              02-26-84
;*****************************************************************************
;
START   EQU     42096   ;DECIMAL LOCATION OF SUB-ROUTINE
ENDTAB  EQU     0FFH    ;END OF TABLE MARKER
;
;   ERROR CODES
NOSTA   EQU     '1'     ;NO SUCH STATE FOUND
WRZIP   EQU     '2'     ;BAD ZIP - DOES NOT MATCH STATE
;
; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;
        ORG     START

BEGIN:  INX     H       ; skip length byte of passed variable
        MOV     D,M     ; put state in DE
        INX     H
        MOV     E,M     ; and save a pointer
        SHLD    ZIPPNT  ; ...to the ZIP bytes


;  First try to locate the state in the table

        LXI     B,8     ; bumper (each table entry is 8 bytes)
        LXI     H,TABLE ; point to first table entry

LOOP1:  MOV     A,M     ; get table character
        CPI     ENDTAB  ; see if the end of the table has
        JZ      ERROR1  ; ... been found and error out if true
        CMP     D       ; Otherwise try to match first state character
        JZ      MAYBY   ; If first character matches we MAY have out state
        DAD     B       ; Otherwise BUMP the table pointer to the next
        JMP     LOOP1   ; entry and try again, and again, and again.....


; Here if first state character matches so try for second match

MAYBY   INX     H       ; Fetch second state character
        MOV     A,M
        CMP     E       ; Check it against passed variable
```

```
        JZ      MATCH   ; ZERO  will be set if valid state was found
        DCX     H       ; Otherwise back up one to keep BUMPER
        DAD     B       ; ...in SYNC and BUMP to next table entry.
        JMP     LOOP1   ; Try again, and again, and again...

;  Here when valid state was found so check if ZIP is within range

MATCH:  PUSH    H       ; Save pointer to TABLE ZIPS
        LHLD    ZIPPNT  ; HL points to passed ZIP
        MOV     B,H     ; BC now points to passed ZIP
        MOV     C,L     ;...and
        POP     H       ;......HL points to TABLE ZIPS

; First check if passed ZIP is smaller than first TABLE ZIP
        CALL    RANGE   ; Check first digit
        JC      ERROR2  ; Carry set if too small
        CALL    RANGE   ; Check second digit
        JC      ERROR2  ; Carry set if too small
        CALL    RANGE   ; Check last digit ONLY if. . .
        JC      ERROR2  ; . . . first two were exact matches

        DCX     B       ; Keep bumper is sync
MATCH1  DCX     B
        DCX     B

        PUSH    B       ; Swap compared elements
        PUSH    H       ; so now we check for upper range
        POP     B
        POP     H

* UPPER LIMIT CHECK

        CALL    RANGE   ; Check first digit
        JC      ERROR2  ; Error out if too large
        CALL    RANGE   ; Check second digit and error
        JC      ERROR2  ; ...out if it is too large
        RNZ             ; If zero then test is passed so return
                        ; ...to calling program
        CALL    RANGE   ; Check last digit ONLY if . . .
        JC      ERROR2  ; . . .first two were exact matches

        RET             ; RETURN to calling program all OK


* RANGE CHECKER
RANGE:  INX     B       ; Compare NEXT byte
        INX     H
        LDAX    B       ; Set CARRY if out of range
        CMP     M
        RET


; ERRORHANDLERS

ERROR1: LHLD    ZIPPNT
        MVI     A,'1'
        JMP     ERROR

ERROR2: LHLD    ZIPPNT
        MVI     A,'2'

* REPLACE VARIABLE WITH ERROR MESSAGE
ERROR:  LHLD    ZIPPNT
        DCX     H
        MVI     M,'E'
```

(Continued on next page)

dBASE II Techniques Listing (Listing Continued, text begins on page 28)

```
        INX     H
        MVI     M,'R'   ;AND NOW ERR
        INX     H
        MVI     M,'R'
        INX     H
        MVI     M,' '
        INX     H
        MOV     M,A
        RET             ; Return to calling program
                        ; ...with passed variable set to ERROR
ZIPPNT: DS      2

; The following info is copied from the US Zipcode book.
; Small possessions and islands are not included but may be added.

TABLE:
        DB      'AL350369'
        DB      'AK995999'
        DB      'AZ850865'
        DB      'AR716729'
        DB      'CA900961'
        DB      'CO800816'
        DB      'CT060069'
        DB      'DE197199'
        DB      'DC200205'
        DB      'FL320339'
        DB      'GA300319'
        DB      'HI967968'
        DB      'ID932938'
        DB      'IL600629'
        DB      'IN460479'
        DB      'IA500528'
        DB      'KS660679'
        DB      'KY400427'
        DB      'LA700714'
        DB      'ME039049'
        DB      'MD206219'
        DB      'MA010027'
        DB      'MI480499'
        DB      'MN550567'
        DB      'MS386397'
        DB      'MO630658'
        DB      'MT590599'
        DB      'NE680693'
        DB      'NV890898'
        DB      'NH030038'
        DB      'NJ070089'
        DB      'NM870884'
        DB      'NY100149'
        DB      'NC270289'
        DB      'ND580588'
        DB      'OH430458'
        DB      'OK730749'
        DB      'OR970979'
        DB      'PA150196'
        DB      'RI028029'
        DB      'SC290299'
        DB      'SD570577'
        DB      'TN370385'
        DB      'TX750799'
        DB      'UT840847'
        DB      'VT050059'
        DB      'VA220246'
        DB      'WA980994'
        DB      'WV247268'
        DB      'WI530549'
        DB      'WY820831'

        DB      ENDTAB

        END
```

End Listings

First Chinese Forth
A Double-Headed Approach

by Timothy Huang

If the computer had been invented in China, what problems would English-speaking people have to surmount in order to use it? To learn computers, a person who comes from an Oriental culture must overcome at least two major obstacles. The first is the natural language barrier and the second is the computer language barrier. Whenever I read an article on computers that is written in English, I first have to understand the English literal meaning, then the technical aspects of the article. Even though I have had many years of English education, I still have trouble now and then. This difficulty will diminish with time, but it will never disappear totally.

With this kind of painful experience in mind (would you guys give me a break and publish articles in plain and simple English?), I started a Chinese Forth computer project, which should be completed by the time this article is published. I primarily wanted to provide a computer system that is more easily accessible to my countrymen, the Chinese. After all, we make up more than a quarter of the world's population: over 1,000,000,000 people. Since the beauty of Forth has been established time and time again, it was, of course, my first choice without reservation. The whole project was written in Dai-E Systems' Forth Level II in both Chinese and English.

The significance of Chinese Forth (or any Chinese language computer system) is obvious. In Taiwan, as well as other Chinese communities, English is a mandatory second language in all schools. But no matter how well a Chinese person masters the English language, Chinese remains the first choice. Computer languages are only software tools invented to enable us to communicate with hardware. The closer this form of expression is to one's native language, the more comfortably the user can proceed, and with much better results.

Many concepts expressed *in* Chinese have no English equivalent, and vice versa. This is one reason why translation is a difficult problem. Since the computer was

"My main objective is to provide the Chinese people with a simple but powerful computer language."

Dai-E Systems, Inc.

HEX	B 8765 / B 4321	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111
0	0000	128	144	SP 160	0 176	@ 192	ㄧ 208	ˊ 224	˙ 240
1	0001	129	145	! 161	1 177	ㄅ 193	ㄕ 209	ㄱ 225	241
2	0010	130	146	" 162	2 178	ㄆ 194	ㄖ 210	ㄲ 226	242
3	0011	131	147	# 163	3 179	ㄇ 195	ㄗ 211	ㄳ 227	243
4	0100	132	148	$ 164	4 180	ㄈ 196	ㄘ 212	ㄴ 228	244
5	0101	133	149	% 165	5 181	ㄉ 197	ㄙ 213	ㄵ 229	245
6	0110	134	150	& 166	6 182	ㄊ 198	ㄚ 214	ㄶ 230	246
7	0111	135	151	' 167	7 183	ㄋ 199	ㄛ 215	ㄷ 231	247
8	1000	136	152	(168	8 184	ㄌ 200	ㄜ 216	ㄸ 232	248
9	1001	137	153	) 169	9 185	ㄍ 201	ㄝ 217	ㄹ 233	249
A	1010	138	154	* 170	: 186	ㄎ 202	ㄞ 218	ㄺ 234	250
B	1011	139	155	+ 171	; 187	ㄏ 203	ㄟ 219	ㄻ 235	251
C	1100	140	156	, 172	< 188	ㄐ 204	ㄠ 220	ㄼ 236	252
D	1101	141	157	- 173	= 189	ㄑ 205	ㄡ 221	ㄽ 237	253
E	1110	142	158	. 174	> 190	ㄒ 206	ㄢ 222	ㄾ 238	254
F	1111	143	159	/ 175	? 191	ㄓ 207	ㄧㄢ 223	ㄿ 239	255

Phonetic Characters Tone Characters

Table I.

Chinese Phonetic Character Chart

Timothy Huang, Dai-E Systems Inc., 29783 Town Center Loop West, P. O. Box 790, Wilsonville, OR 97070.

born in the United States, most of the current Forth-related subjects are expressed in English: SPACE, SPACES, BLANK, BLANKS, etc. For the average American, these constructions are natural, but the Chinese do not express plurals in this manner; they have a different philosophy, a different way of thinking, and different forms of expression.

Forth is expressed nicely in English, so why not in the oldest language – Chinese? For the Chinese to realize real computer power (e.g., Forth power), it should be available in their own language.

My main objective is to provide the Chinese people with a simple but powerful computer language. I want to remove the English language barrier so that even Chinese children can use Forth. Furthermore, I would like to see the impact on Forth of exposure to the Chinese culture and language. I hope to see great and positive contributions and believe that only Forth can absorb this challenge. Other less conceptual languages will have a difficult time, because the Chinese philosophy is very abstract.

Double (Multiple) Names

A Forth word with two different names can be made easily by:

```
: PUD DUP;
    or
:FI [COMPILE] IF ; IMMEDIATE
```

but there are many deficiencies in this method. For example, the execution speed will be hampered because the new word(s) must execute at least one extra cycle of the inner interpreter.

In his article entitled "Turtle Talk," Glenn Tenney described a defining word, ALIAS, that stores the old word's cfa into the new word's pfa.[1] The new word (alias) is created as an immediate word: when alias is executed, the old word (parent) must be executed. This approach solved some problems but also created its own. In our case, not all Chinese words are immediate.

Since the implementation of Dai-E Systems' Forth Level II (83 Standard) uses a separate header and body approach, we can achieve the double (multiple) name easily by simply creating a new header with a pointer pointing to the same body of the old word. The separated heads concept was first proposed and implemented by Klans Schleisiek.[2] I'm not arguing the respective merits of continuous or separated heads/bodies, but only pointing out how I did it. Using the separated heads scheme, the memory map of a word is shown in Figure 1 (below).

The pointer in the above structure is the key, since nothing forbids more than one head pointing to the same body. The only no-no is to have one head pointing to many bodies, which would result in a very interesting Forth system. Equipped with this understanding, we can make

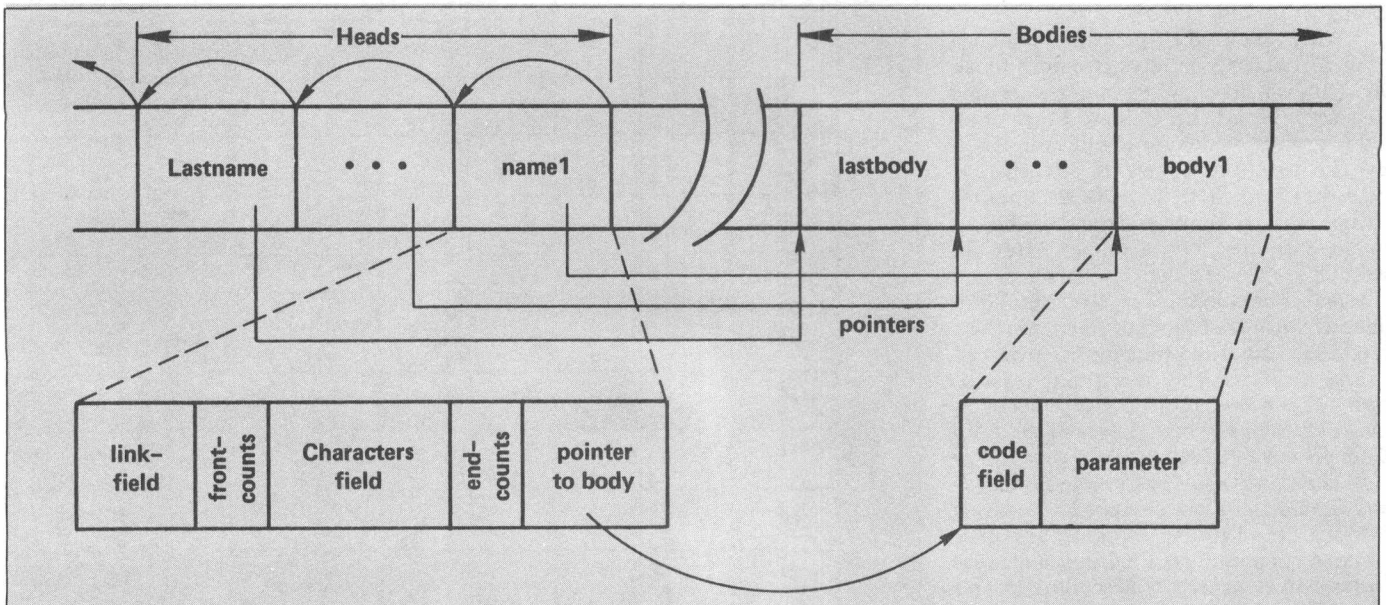


Figure 1. Separated Heads/Bodies Structure

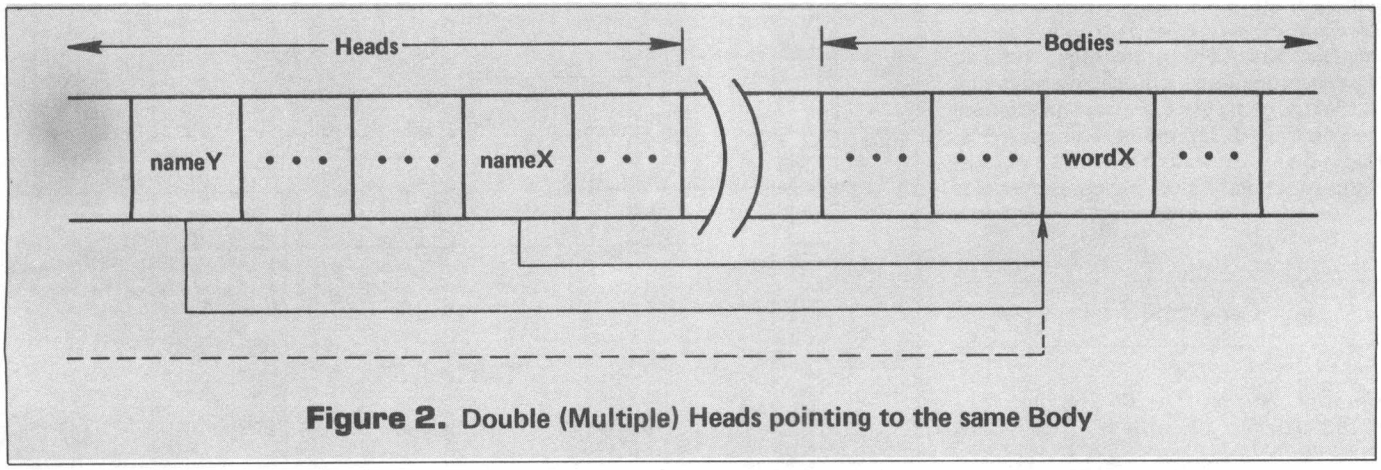


Figure 2. Double (Multiple) Heads pointing to the same Body

double (multiple) headers as shown in Figure 2 (page 33).

To accomplish this, we have to store the old pointer from the old word into the pointer location of the new name, as described in Figure 3 (below). The word C: should be used to generate new names. For example:

C: PUD DUP ;

C: FI IF ; IMMEDIATE

The new words (actually new names) could be either immediate or non-immediate words depending on whether we issue IMMEDIATE after the definition.

This simple approach not only eliminates one extra nesting level but also gives us the choice of indicating the immediate or non-immediate nature of a word. With this trick we can have one word behaving differently depending on which header we use; i.e., one name is immediate and the other is non-immediate.

Chinese Forth

Chinese characters are ideographic symbols. Each one is an individual entity, and there are more than 50,000 of them. Because it is impossible to make an ASCII table to include all of them, we took a simpler approach and used the phonetic characters rather than the actual ideograms. There are 37 phonetic symbols and five tone symbols, as shown in Table I (page 32). With a proper combination of at least one, but no more than three, phonetic symbols and one tone symbol, all the ideographic characters can be expressed.

These phonetic characters can be placed into the higher portion of the ASCII table with the most significant bit on. This phonetic character set is called the "BER PER MER FER"(ㄅ ㄆ ㄇ ㄈ). This is the first thing learned when a person enters a formal education institution. The phonetic characters are learned first, then the pronunciation, then the ideographic characters. Eventually, phonetic characters (pronunciation) are no longer needed alongside each ideographic character.

As mentioned earlier, more than 50,000 ideographic characters exist now; however, there are only about 2,000 legal pronunciations (combinations of sounds and tones) for them. Thus, the homonym (different character with identical pronunciation) is a very annoying problem. Theoretically, each pronunciation can apply to about 25 different ideographic characters. However, the Chinese created a clever method for alleviating this confusion: they used multiple characters to form a phrase as often as possible rather than single characters. With phrases of at least two characters, the chance of a mix-up was greatly reduced. The telephone directory service of the Taiwan Telephone and Telegraph Company is using this method in conjunction with their computer systems with great success.

For this reason, I decided to avoid single-character phonetic names for the Chinese Forth words. Each English Forth word was translated to a Chinese phrase of more than two characters. However, the math operators remained the same. Certain English characters used in the name fields were also preserved, such as " ;" " " " " " (" and so on.

Table II (page 36) lists all the required words in the 83 Standard. The first column is the original English, the second the equivalent Chinese ideographic characters, the third the phonetic names used in our system. Within the phonetic names, the first tone character, which is usually the blank character, was changed to a high raised "-" character to avoid the space character limitation of Forth names.

By using the phonetic characters, even though I cannot see the ideographic characters, I can pronounce the program and thus have no problem whatsoever in understanding what I have written. This computerized solution solves the 50,000-character ASCII table problem. Besides, this reflects the hope that Forth, a written language, might be a spoken language as well. Otherwise, why is there a pronunciation for words such as @ (fetch) or ! (store)? Chinese Forth not only can be written but also can be verbalized (spoken).

The other interesting thing regarding our Chinese Forth is, while decompiling the English-origin word, one may see half English and half Chinese, depending on how many English words were translated into their Chinese equivalents. This is because we utilized the double-header approach mentioned earlier.

```
: C:    (S -------- )      \ create double header
create                     \ make new name field
   '                       \ find the cfa of the old word
heads                      \ set dictionary pointer to header
here 2-                    \ into the new pointer address
   !                       \ store old cfa into new pointer address
bodies                     \ reset dictionary pointer to body portion
   ;
```

Figure 3.
Double (Multiple) Header Code

一 九 八 三 年 電 腦 大 展
達 意 資 訊 公 司
正 式 展 出
中 文 電 腦 系 統
敬 請 多 多 指 教

Figure 4.
Characters printed on Axiom IMP4

References

1. Glenn S. Tenney. "Turtle Talk." *FORML*, 1981, pp. 521-542.
2. Klans Schleisiek. "Separated Heads." *FORTH DIMENSIONS*, Vol. II/5, p. 147.

The Dai-E Chinese Computer System includes a Victor 9000 microcomputer with 512K of RAM, Axiom IMP4 printer, and 5,000 of the most commonly used Chinese characters encoded to conform to the communication protocol selected by the U. S. Library of Congress, CCCII – Chinese Character Code for Information Interchange. In addition to their Forth system, they have a Chinese word processor. They should be able to use their system shortly on the British-made Apricot computer. They are also in the process of bringing the system up on other computers. – Ed. **DDJ**

(Table II begins on page 36)

(( )) Indicates Chinese Pronunciation only

English Nucleus Layer	Chinese Character Name		Chinese Forth Phonetic Name
!		(儲存)	!
*		(相乘)	*
*/		(相除)	*/
+		(相加)	+
+!		(加存)	+!
-		(相減)	-
/		(相除)	/
/MOD		(餘除)	/MOD
0<		(小於零)	0<
0=		(等於零)	0=
1+		(壹加)	1+
1-		(壹減)	1-
2+		(貳加)	2+
2-		(貳減)	2-
2/		(貳除)	2/
<		(小於)	<
=		(等於)	=
>		(大於)	>
>R		(到回疊)	>R
?DUP	?重覆		?ㄔㄨㄥˊㄈㄨˋ
@		(取得)	@
ABS	絕對值		ㄐㄩㄝˊㄉㄨㄟˋㄓˊ
AND	聯與		ㄌㄧㄢˊㄩˇ
C!		(存位)	C!
C@		(取位)	C@
CMOVE	移位元		ㄧˊㄨㄟˋㄩㄢˊ
CMOVE>	反移位		ㄈㄢˇㄧˊㄨㄟˋ
COUNT	計算		ㄐㄧˋㄙㄨㄢˋ
D+		(雙加)	D+
D<		(雙小於)	D<

Table II.
Chinese Forth-83 Standard Words

DEPTH	深度		ㄕㄣ ㄉㄨˋ
DNEGATE	雙負值		ㄕㄨㄤ ㄈㄨˋ ㄓˊ
DROP	丟棄		ㄉㄧㄡ ㄑㄧˋ
DUP	重複		ㄔㄨㄥˊ ㄈㄨˋ
EXECUTE	執行		ㄓˊ ㄒㄧㄥˊ
EXIT	出來		ㄔㄨ ㄌㄞˊ
FILL	填充		ㄊㄧㄢˊ ㄔㄨㄥ
I		(次序)	I
J		(外序)	J
MAX	極大		ㄐㄧˊ ㄉㄚˋ
MIN	極小		ㄐㄧˊ ㄒㄧㄠˇ
MOD		(除餘)	MOD
NEGATE	負值		ㄈㄨˋ ㄓˊ
NOT	非反		ㄈㄟ ㄈㄢˇ
OR	或者		ㄏㄨㄛˋ ㄓㄜˇ
OVER	轉疊		ㄓㄨㄢˇ ㄉㄧㄝˊ
PICK	選疊		ㄒㄩㄢˇ ㄉㄧㄝˊ
R>		(到回疊)	R>
R@		(複回疊)	R@
ROLL	混轉		ㄏㄨㄣˊ ㄓㄨㄢˇ
ROT	翻轉		ㄈㄨㄢ ㄓㄨㄢˇ
SWAP	對調		ㄉㄨㄟˋ ㄉㄧㄠˋ
U<		(正小於)	U<
UM*		(正混乘)	UM*
UM/MOD		(正混除餘)	UM/MOD
XOR	僅或		ㄒㄧㄣˇ ㄏㄨㄛˋ

Device Layer

BLOCK	區段	ㄑㄩ ㄉㄨㄢˋ
BUFFER	緩衝	ㄏㄨㄢˇ ㄔㄨㄥ
CR	換行	ㄏㄨㄢˋ ㄏㄤˊ
EMIT	放送	ㄈㄤˋ ㄙㄨㄥˋ
EXPECT	等候	ㄉㄥˇ ㄏㄡˋ
FLUSH	清存	ㄑㄧㄥ ㄘㄨㄣˊ
KEY	字鍵	ㄗˋ ㄐㄧㄢˋ

(Continued on next page)

(Continued from page 37)

SAVE-BUFFERS	存緩衝區	ㄙㄨㄙˊㄈㄨㄋˇㄎㄨㄢˉㄑㄩ
SPACE	空格	ㄎㄨㄥˉㄍㄜˊ
SPACES	空間	ㄎㄨㄥˉㄐㄧㄢ
TYPE	印字	ㄧㄣˋㄗˋ
UPDATE	標新	ㄅㄧㄠˉㄒㄧㄣ

Interpreter Layer

#		(換數)	#
#>		(停換)	#>
#S		(全換)	#S
#TIB	#終入區	)	#ㄓㄨㄥˉㄖㄨˋㄑㄩ
'		(提取)	'
(		(左括弧)	(
-TRAILING	去尾		ㄑㄩˋㄨㄟˇ
.		(點)	.
.(		(點括弧)	.(
<#		(始換)	<#
>BODY	〉身體		>ㄈㄉˉㄊㄧˇ
>IN	〉入標		>ㄖㄨˋㄅㄧㄠ
ABORT	結束		ㄐㄧㄝˊㄙㄨˋ
BASE	基數		ㄐㄧˉㄈㄨˋ
BLK	表數		ㄅㄧㄠˇㄈㄨˋ
CONVERT	變換		ㄅㄧㄢˋㄏㄨㄢˋ
DECIMAL	十進位		ㄕˊㄐㄧㄣˋㄨㄟˋ
DEFINITIONS	添詞		ㄊㄧㄢˉㄘˊ
FIND	尋找		ㄒㄩㄣˊㄓㄠˇ
FORGET	忘記		ㄨㄤˋㄐㄧˋ
FORTH	福式		ㄈㄨˊㄈㄛˋ
FORTH-83	福式 - 83		ㄈㄨˊㄈㄛˋ—83
HERE	這兒		ㄓㄜˋㄦˊ
HOLD	包函		ㄅㄠˉㄏㄢˊ
LOAD	譯入		ㄧˋㄖㄨˋ
PAD	填補區		ㄊㄧㄢˊㄅㄨˇㄑㄩ
QUIT	終止		ㄓㄨㄥˉㄓˇ

SIGN	符號		ㄈㄨˊㄏㄠˋ
SPAN	延申		ㄧㄢˊㄕㄣ
TIB	終入區		ㄓㄨㄥ ㄖㄨˋㄑㄩ
U.		(正點)	U.
WORD	詞字		ㄘˊㄗˋ

Compiler Layer

+LOOP	+轉合		+ㄓㄨㄢˇㄏㄜˊ
,		(入位)	,
."		(示字)	."
:		(開始定美)	:
;		(定義結束)	;
ABORT"	結束"		ㄐㄧㄝˊㄙㄨˋ"
ALLOT	保留		ㄅㄠˇㄌㄧㄡˊ
BEGIN	開始		ㄎㄞ ㄕˇ
COMPILE	編碼		ㄅㄧㄢ ㄇㄚˇ
CONSTANT	常數		ㄔㄤˊ ㄕㄨˋ
CREATE	創造		ㄔㄨㄤˋ ㄗㄠˋ
DO	起承		ㄑㄧˇ ㄔㄥˊ
DOES>	操作〉		ㄘㄠ ㄗㄨㄛˋ〉
ELSE	不然		ㄅㄨˋ ㄖㄢˊ
IF	假如		ㄐㄧㄚˇ ㄖㄨˊ
IMMEDIATE	立即		ㄌㄧˋ ㄐㄧˊ
LEAVE	離開		ㄌㄧˊ ㄎㄞ
LITERAL	編數		ㄅㄧㄢ ㄕㄨˋ
LOOP	轉合		ㄓㄨㄢˇㄏㄜˊ
REPEAT	重來		ㄔㄨㄥˊ ㄌㄞˊ
STATE	狀態		ㄓㄨㄤˋ ㄊㄞˋ
THEN	否則		ㄈㄡˇ ㄗㄜˊ
UNTIL	直到		ㄓˊ ㄉㄠˋ
VARIABLE	變數		ㄅㄧㄢˋ ㄕㄨˋ
VOCABULARY	字彙		ㄗˋ ㄏㄨㄟˋ
WHILE	正當		ㄓㄥˋ ㄉㄤ
[		(換態)	[
[']		(立即提)	[']
[COMPILE]	〔編碼〕		[ㄅㄧㄢ ㄇㄚˇ]
]		(回態)	

End Table

cc A Driver for a Small-C Programming System

Dr. Schreiner was kind enough to provide us with two programs for use with the Small-C compiler: cc, *presented here, and* p, *a stand-alone Small-C preprocessor. References to* p, *in the following discussion are to that preprocessor, which we will publish next month. – Ed.*

Once you make extensive use of a programming system consisting of a compiler, assembler, and linker, you find yourself either typing a lot of commands or using the CP/M SUBMIT facility quite a bit. The latter, however, is not very flexible. It will run unconditionally whatever commands the batch file dictates; those, regardless of argument substitution, may be more or less than what you intended.

A language like C encourages separate compilation and program composition from various source files. A combination like Jim Hendrix's Small-C compiler (*DDJ*, December 1982) and MicroSoft's relocating assembler and linking loader makes it quite attractive to compile as little as possible during program development. When you add a separate preprocessor for Small C and attempt to eliminate intermediate files, you find yourself typing a lot of (almost) identical commands each time you want to preprocess, compile, erase, assemble, erase, link, and so on.

cc (Listing One, page 44) is a program patterned after Dennis Ritchie's *cc* command in the Unix* system. It accepts options and file specifications and, through CP/M's SUBMIT feature, arranges for the proper amount of preprocessing, compiling, assembly, and loading. Essentially you type what files ought to be processed to construct a program, then *cc* prepares a SUBMIT file and persuades CP/M to execute it.

Features

cc is based on a runtime support that passes arguments to the main program. It expects to be called as follows:

cc (option) . . . file . . .

You may specify options in any order. They are generally cumulative. The following options are available:

-c	Compile only; do not execute the linker
-p	Preprocess only
-s	Preprocess and compile only
-o filename	Output of the linker is "filename"

Other options are passed on to the relevant processor, mostly to the Small-C preprocessor. *p* accepts the following:

-d name (=value)	Define a symbolic name
-e	Suppress position stamps
-i drive	Search for file inclusion
-u name	Undefine a symbolic name

So that *p* can be used prior to Hendrix's compiler, the *-e* option is included automatically.

Files are passed on to the appropriate processors. The more general files are processed first. Intermediate files, named after the original source files, are erased once they are no longer needed; objects, however, are retained. In the absence of the *-o* option, the resulting load module will be named after the first of the more general source files. *cc* uses the following filename extensions:

.c	Preprocessor source file
.i	Compiler source file
.mac	Macro assembler source file
.rel	Linker source file or library

If an extension is not present or is unrecognizeable, the file is passed only to the linker. It is thus quite simple to pass libraries. Files are passed to the linker within each source file category in the order specified; the more general source file category, however, is passed first. This rarely produces conflicts.

The options are clearly patterned after the Unix system. The main program expects to receive pointers to the various options as a vector "argv." The number of such options, including (theoretically) the program name as first option, is also passed as an integer "argc." Options must precede the filenames.

Examples

A few sample calls might illustrate just what *cc* does. The first call will construct the *cc* command itself:

```
B>cc cc.c
```

On my system I have the shortest possible names for the language processors, and I keep most tools on disk a:. Here the following commands would be issued:

```
A>b:p-e b:cc.c >b:cc.i
A>c b:cc.i >b:cc.mac
A>era b:cc.i
A>m =b:cc
A>era b:cc.mac
A>l b:cc/n/e,b:cc,c/s
A>b:
```

Assume that submit.c is compiled separately:

```
B>cc -c submit.c
A>b:p -e b:submit.c >b:submit.i
A>c b:submit.i >b:submit.mac
A>era b:submit.i
A>m =b:submit
A>era b:submit.mac
A>b:
```

Subsequently we can compile cc.c and link it as follows:

```
B>cc submit cc.c
A>b:p -e b:cc.c >b:cc.i
A>c b:cc.i >b:cc.mac
A>era b:cc.i
A>m =b:cc
A>era b:cc.mac
A>l b:cc/n/e,b:cc,submit,c/s
A>b:
```

It should be clear why such a driver program might be useful not only for Small C.

Implementation

The system driver basically has the following structure:

- obtain and save options, initialize
- obtain, test, and save files
- *for* each category of source files run preprocessor, compiler, assembler as required
- run linker as required

Most of this is accomplished by main( ); one subroutine for each processor handles the problem of issuing the actual commands. This arrangement makes the system easy to adapt to other processors.

File and option lists must be circular so that they can be traversed easily in input order. Push routines maintain the lists. A list element consists of a pointer to the next list element, followed by

by Axel Schreiner

Axel T. Schreiner, Universität Ulm, Sektion Meth. d. Informatik, Oberer Eselsberg, 7900 Ulm (Doran) West Germany.

**Unix is a trademark of Bell Laboratories.*

the string that constitutes the actual value stored. The list header points to the last element entered, which in turn points to the first element, and the circular list continues on to the last element again. The list need not be marked since you always start at the list header and thus know when you have traversed the list once.

For reasons explained below, filenames must be fully specified. If an explicit drive name is not part of the file specification, pushf( ) will add the current drive name.

The kind( ) function analyzes a file specification to determine the source category based on the file extension. All "unknown" extensions are considered to be object specifications. This allows linker libraries to pass through correctly. All other files must be existing source files. To detect trivial errors early, kind( ) tests all source files for existence by opening them for reading.

submit( )

The most interesting aspect of *cc* clearly is its use of the CP/M SUBMIT feature. During a warm start on drive a:, the CP/M console command processor (CCP) checks if a file $$$.sub exists. If it does, it is expected to contain one CCP command line (preceded by its length) per CP/M sector. The CCP takes the *last* sector from the file, truncates the file by one sector, and issues the command.

To use this feature, you need to store all command lines as a stack then write them, appropriately formatted, into the file. You also need to take care that, once the driver program exits, drive a: is selected and a warm start is performed.

The submit( ) function (Listing Two, page 52) handles all of this. If it is called with a string argument, it will stack the string (using a dynamic memory allocation function that is part of my runtime support). If it is called with a NULL argument, it will write the stack (i.e., the commands in reverse order) into the file, force the CCP to select drive a: (and user area 0) by clearing the byte at location 4, and terminate program execution through a warm start.

submit( ) issues a command to (re)-select the current drive, so that once the batch stream is completely processed the current drive is selected again. Nevertheless, during batch execution, drive a: must be selected. This is why pushf( ) fully specifies each file.

submit( ) actually appends to the end of the batch file processed by the CCP. This has a desirable effect in that submit( ) can be used from *within* a batch stream – something that is lacking in the CP/M SUBMIT utility. (It is not very difficult to position a CP/M file to end of file, especially on a sector boundary.) A simple test program (Listing Three, page 55) demonstrates this feature:

```
B>x a "b c' "'d e' f" g
a
    A>b:x b c
    b
        A>b:x c
        c
        A>a:
    A>b:x 'd e' f
    d e
        A>b:x f
        f
        A>a:
    A>b:x g
    g
    A>b:
```

Quotes and double quotes can be used to hide white space within command arguments. The output shown above has been indented to show the nesting of batch streams.

Installation

Getting *cc* to run on your system might be tricky. Basically you need to decide where your processors and standard libraries reside and how they ought to be called. Also, *cc* relies on my own runtime support, which is heavily oriented towards the Unix environment.

You probably should consult Kernighan and Ritchie's book *The C Programming Language* (Prentice Hall, 1978) to learn about all the routines that are mentioned "extern" at the beginning of the program. Most of them are quite simple to construct. It is essential, however, that you provide a memory allocator, calloc( ), that is reasonably stable. I am using a scheme where memory above the load module and below the stack is managed by a list of words, each word pointing to the next. The low bit in each such word indicates if the area past that word and up to the next one is allocated or free.

I have had access to Chapter 17 of Hendrix's book on Small C, describing his runtime support. [*See this month and last month for Payne's and Hendrix's new library – Ed.*] While I did not have access to the runtime support itself and therefore could not test it, I believe that installing *cc* should be quite simple. The following probably must be done:

FILE	should be defined as *int*.
_drive( )	needs to access BDOS function 25.
fseek(. . , 10)	is Hendrix's cseek(. . , 2).
rindex( )	is Hendrix's strrchr( ).

You can replace _bputchar( ) by storing the relevant characters in a buffer of length 128 and using Hendrix's write( ) to emit the buffer. The file pointer _cfp then is not needed.

I process the arguments to main( ) directly. Depending on the actual implementation, you might have to use Hendrix's function getarg( ). I am assuming that the storage allocator, calloc( )/cfree( ), supports random order release of memory.

Two runtime routines deserve special mention:

```
fseek(fp, 0, 10)
```

will position the file indicated by "fp" to the end of the last allocated CP/M sector. (In Unix the third argument for fseek( ) must be 0, 1, or 2, indicating positioning relative to the beginning of the file, the present position, and the end of the file. Additionally, I permit 8, 9, and 10, indicating sector positioning.)

```
_bputchar(ch)
```

will emit the character "ch" to the file currently indicated by "_cfp" without interpreting tab, return, or other characters. Since submit( ) must write CP/M sectors containing binary length information, use of this very internal routine of the runtime support is necessary. _bputchar( ) returns EOF on end of file, e.g., when the relevant disk overflows.

Deciding where your processors live and how they ought to be called is your own problem. I have a CP/M system with two 200K floppies; one of these (barely) contains Small C, the MicroSoft assembler and linker, a text editor, and my runtime library. Unless I trade the text editor for the preprocessor, I have to call the preprocessor from the second disk.

Notice that the SUBMIT file must be on the a: disk and that this disk must be selected while this file is processed. My processors therefore all sit on this disk, with enough room left over for the SUBMIT file.

Constructing the proper calls is relatively simple; if you use other systems, you may have to add a small amount of code to the program.

Part of this work was done during a sabbatical spent at the University of Illinois; in particular, the Small-C system was obtained from "UseNet."

■■J

(Listing begins on page 44)

CC (Text begins on page 40)

Listing One

```
/*
 *      cc - small C compiler driver
 *      ats 5/83
 */

/*
 *      define...
 *
 *      c80     to drive Software Toolworks C80
 */

char    usage[] =
        "cc [-c|p|s] [-d n[=v]] [-e] [-i d:] [-o task] [-u n] files";

#define Csource 1       /* results of kind() */
#define Msource 2
#define Psource 3

#define Cfile   ".i"    /* for these extensions */
#define Pfile   ".c"
#define Mfile   ".mac"

#define C       "c"     /* for these processors */
#define P       "b:p"
#define M       "m"
#define L       "l"

#define CLIB    "c/s"   /* and this library search */

/*
 *      "FEATURES:"
 *
 *              If the same file name is specified with different
 *              extensions, the most general source is processed...
 *              ...and the others will fail, but be linked several times!
 */

#include <stdio.h>

/*
 *      i/o header file

#define FILE    ???     type to represent files (used as FILE*)
#define stdin   ???     pre-opened standard input file
#define stdout  ???     pre-opened standard output file
#define stderr  ???     pre-opened diagnostic output file
#define NULL    0       null pointer, false
#define EOF     ???     end of file indication

 */

/*
 *      special data type
 */

#define LIST        int             /* list of word or string values */

#define l_next(x)   (*(x))          /* -> next element */
#define l_str(x)    ((x)+1)         /* -> string value */
#define l_first(x)  (*(x)+2)        /* -> first string value in circular list */
#define sz_STR(s)   (3+strlen(s))   /* size of string list element */
#define sz_FILE(s)  (5+strlen(s))   /* size of file (+ drive) element */
```

```
/*
 *      runtime support routines
 */

extern  _drive();       /* BDOS function 25: current drive number */
        calloc();       /* (n,l) return NULL or -) n elements of length l */
        cfree();        /* (p) free area at p, returned by calloc() */
        exit();         /* terminate program execution */
        fputs();        /* (s,f) write string s on file f */
        freopen();      /* (n,m,f) return NULL or file descriptor f,
                           closed if necessary, and reopened to read
                           (m == "r"), write ("w"), or append ("a") */
        rindex();       /* (s,c) find c in string s end to front, return
                           NULL or ->c in s; '\0' is always found */
        strcat();       /* (a,b) copy string b beyond string a */
        strcmp();       /* (a,b) <, ==, > 0 as string a is <, ==, > string b */
        strcpy();       /* (a,b) copy string b to string a */
        strlen();       /* (s) return number of characters in string s */
        submit();       /* (s) add string s to command list; (NULL) submit */

/*
 *      global data
 */

char    buf[126],       /* to submit (length unchecked...) */
        drive[] = "?:", /* current drive name */
        pflag,          /* run only preprocessor */
        cflag,          /* run only compiler */
        sflag;          /* run assembler */

LIST    *task,          /* task name (-o option) only */
        *popt,          /* p option list */
        *parg,          /* p argument list */
        *carg,          /* c argument list */
        *marg,          /* m argument list */
        *larg;          /* l argument list */

main(argc, argv)
        int argc;
        int *argv;
{       char *cp;
        LIST *arg;

        /* remember current drive */
        drive[0] = _drive() + 'a';

        /* default preprocessor options */
        popt = pushs(popt, "-e");

        /* record options and flags */
        while (--argc)
        {       cp = *++argv;
                /* options must precede files */
                if (*cp != '-')
                        break;
                /* dispatch and record, accept values attached or separate */
                switch(cp[1]) {
                case 'c':               /* -c           run p c m */
                        cflag = 1;
                        break;
                case 'd':               /* -d n[=v]     p option */
                case 'i':               /* -i d:        p option */
                case 'u':               /* -u n         p option */
                        popt = pushs(popt, *argv);
                        if (cp[2])
                                break;
```

(Continued on next page)

Listing One

```
                        if (--argc == 0)
                                goto error;
                        popt = pushs(popt, *++argv);
                        break;
                case 'e':               /* -e           p option */
                        popt = pushs(popt, *argv);
                        break;
                case 'o':               /* -o task      name task image */
                        if (cp[2])
                                task = pushf(task, cp+2);
                        else if (--argc == 0)
                                goto error;
                        else
                                task = pushf(task, *++argv);
                        break;
                case 'p':               /* -p           run p */
                        pflag = 1;
                        break;
                case 's':               /* -s           run p c */
                        sflag = 1;
                        break;
                default:
                        goto error;
                }
        }

        /* there must be at least one file */
        if (argc == 0)
        {
error:          fputs(usage, stderr);
                exit();
        }

        /* collect files on various lists */
        do
        {       switch(kind(*argv)) {
                case Psource:
                        parg = pushf(parg, *argv);
                        break;
                case Csource:
                        carg = pushf(carg, *argv);
                        break;
                case Msource:
                        marg = pushf(marg, *argv);
                        break;
                default:
                        larg = pushf(larg, *argv);
                }
                ++argv;
        } while (--argc);

        /* run files preprocessor -> compiler -> assembler */
        if (arg = parg)
                do
                {       arg = *arg;
                        run(P, popt, l_str(arg), Pfile, Cfile);
                        if (pflag)
                                continue;
                        run(C, NULL, l_str(arg), Cfile, Mfile);
                        erase(l_str(arg), Cfile);
```

```
                if (sflag)
                        continue;
                asm(l_str(arg));
                erase(l_str(arg), Mfile);
        } while (arg != parg);
if (pflag)
        goto done;

/* run files compiler -) assembler */
if (arg = carg)
        do
        {       arg = *arg;
                run(C, NULL, l_str(arg), Cfile, Mfile);
                if (sflag)
                        continue;
                asm(l_str(arg));
                erase(l_str(arg), Mfile);
        } while (arg != carg);
if (sflag)
        goto done;

/* run files assembler */
if (arg = marg)
        do
        {       arg = *arg;
                asm(l_str(arg));
        } while (arg != marg);
if (cflag)
        goto done;

/* run (MicroSoft) linker */
/* L task/n/e, parg..., carg..., marg..., larg..., CLIB */
/* note that we do not explicitly check the length of this command */

strcpy(buf, L);
strcat(buf, " ");


/* decide on output file name */
if (task)
        strcat(buf, l_str(task));
else if (parg)
        strcat(buf, l_first(parg));
else if (carg)
        strcat(buf, l_first(carg));
else if (marg)
        strcat(buf, l_first(marg));
else if (larg)
        strcat(buf, l_first(larg));
else
{       strcat(buf, drive);
        strcat(buf, "task");
}
strcat(buf, "/n/e");

/* add all modules */
if (arg = parg)
        do
        {       arg = *arg;
                strcat(buf, ",");
                strcat(buf, l_str(arg));
        } while (arg != parg);
```

(Continued on next page)

Listing One

```
        if (arg = carg)
                do
                {       arg = *arg;
                        strcat(buf, ",");
                        strcat(buf, l_str(arg));
                } while (arg != carg);
        if (arg = marg)
                do
                {       arg = *arg;
                        strcat(buf, ",");
                        strcat(buf, l_str(arg));
                } while (arg != marg);
        if (arg = larg)
                do
                {       arg = *arg;
                        strcat(buf, ",");
                        strcat(buf, l_str(arg));
                } while (arg != larg);

        /* add smallC library and submit */
        strcat(buf, ",");
        strcat(buf, CLIB);
        submit(buf);

        /* submit the batch stream */
done:
        submit(NULL);
}

/*
 *      circular list routines
 */
pushs(l, s)             /* attach string to list */
        LIST *l;        /* list */
        char *s;        /* string */
{       LIST *r;        /* new element */

        if ((r = calloc(sz_STR(s), 1)) == NULL)
        {       fputs("no room", stderr);
                exit();
        }
        strcpy(l_str(r), s);
        /* empty list: link first element to itself */
        if (l == NULL)
                return l_next(r) = r;
        /* nonempty list: tie element into it */
        l_next(r) = l_next(l);
        return l_next(l) = r;
}

pushf(l, f)             /* attach file name to list */
        LIST *l;        /* list */
        char *f;        /* file name */
{       LIST *r;        /* new element */

        if ((r = calloc(sz_FILE(f), 1)) == NULL)
        {       fputs("no room", stderr);
                exit();
        }
        if (f[1] != ':')        /* use current drive */
```

```
        {       strcpy(l_str(r), drive);
                strcat(l_str(r), f);
        }
        else                    /* explicit drive */
                strcpy(l_str(r), f);
        if (l == NULL)
                return l_next(r) = r;
        l_next(r) = l_next(l);
        return l_next(l) = r;
}

/*
 *      source file type analysis
 */

kind(f)                 /* determine type of source */
        char *f;        /* file name */
{       char *p;

        if (p = rindex(f, '.'))
                if (strcmp(p, Pfile) == 0)
                {       if (freopen(f, "r", stdin) == NULL)
                                goto badfile;
                        *p = '\0';
                        return Psource;
                }
                else if (strcmp(p, Cfile) == 0)
                {       if (freopen(f, "r", stdin) == NULL)
                                goto badfile;
                        *p = '\0';
                        return Csource;
                }
                else if (strcmp(p, Mfile) == 0)
                {       if (freopen(f, "r", stdin) == NULL)
                                goto badfile;
                        *p = '\0';
                        return Msource;
                }
        return 0;
badfile:
        fputs("cannot open ", stderr);
        fputs(f, stderr);
        exit();
}

/*
 *      routines to produce command calls
 */

run(cmd, opt, fnm, in, out)     /* run smallC task */
                        /* cmd opt... fnm.in > fnm.out */
        char *cmd;      /* command name */
        LIST *opt;      /* option list */
        char *fnm;      /* file name to process */
        char *in;       /* input file extension */
        char *out;      /* output file extension */
{       LIST *p;

        strcpy(buf, cmd);
        if (p = opt)
                do
                {       p = l_next(p);
                        strcat(buf, " ");
                        strcat(buf, l_str(p));
```

(Continued on page 52)

Listing One

```
                } while (p != opt);
        strcat(buf, " ");
        strcat(buf, fnm);
#ifndef c80
        strcat(buf, in);
        strcat(buf, " )");
        strcat(buf, fnm);
        strcat(buf, out);
#endif
        submit(buf);
}

asm(fnm)                /* run (MicroSoft) macro assembler */
                        /* M = fnm */
        char *fnm;      /* file name to process */
{
        strcpy(buf, M);
        strcat(buf, " =");
        strcat(buf, fnm);
        submit(buf);
}

erase(fnm,ext)          /* erase intermediate file */
                        /* era fnm.ext */
        char *fnm;      /* file name */
        char *ext;      /* extension */
{
        strcpy(buf, "era ");
        strcat(buf, fnm);
        strcat(buf, ext);
        submit(buf);
}
```

End Listing One

Listing Two

```
/*
 *      submit() -- submit commands to CP/M batch
 *      ats 5/83
 */

#define DSKBYTE 4       /* BDOS/CCP selected disk, user# */

/*
 *      needed from the i/o header file

#define FILE    ???     type to represent files
#define NULL    0       null pointer, false
#define EOF     ???     end of file indication

 */

/*
 *      runtime support routines
 */
```

```
extern  _bputchar();    /* (c) write char c uninterpreted to FILE *_cfp */
        _cfp;
        _drive();       /* BDOS function 25: current drive number */
        calloc();       /* (n,l) return NULL or -> n elements of length l */
        exit();         /* terminate program execution */
        fclose();       /* (f) close file described by f */
        fopen();        /* (n,m) return NULL or descriptor for file "n"
                           opened to append (m == "a") */
        fseek();        /* (f,p,w) position f to byte or sector p measured
                           according to w */
        strcpy();       /* (a,b) copy string b to string a */
        strlen();       /* (s) return number of characters in string s */

/*
 *      global data
 */

int *_submit;           /* stack of submitted buffers */

submit(s)               /* add command to CP/M job stream */
                        /* returns argument or NULL */
                        /* on NULL argument: submits and exits */
        char * s;       /* command to submit */
{       FILE *fp;       /* need to use block i/o, NOT putc */
        char *cp;
        int *wp, i;

        /* if string argument, save it as a command */
        if (s)
        {       if (strlen(s) ) 126)
                        return NULL;    /* too long */
                if ((wp = calloc(strlen(s)+2+2,1)) == NULL)
                        return NULL;    /* no room */
                *wp = _submit;
                _submit = wp;
                cp = _submit+1;

                *cp = strlen(s);        /* length byte */
                strcpy(cp+1, s);        /* text */
                return s;
        }

        /* if anything to submit - write it to a:$$$.sub */
        if (_submit)
        {       if ((fp = fopen("a:$$$.sub", "a")) == NULL)
                        return NULL;    /* cannot make batch file */
                if (fseek(fp, 0, 10) == -1)
                        goto error;
                _cfp = fp;      /* for _bputchar() */

                /* last command: reselect current drive */
                _bputchar(2);
                _bputchar(_drive()+'a');
                _bputchar(':');
                for (i = 3; i ( 127; ++i)
                        _bputchar(0);
                /* check last byte in sector for overflow */
                if (_bputchar(0))
                        goto error;

                /* other commands from stack */
                do
```

(Continued on next page)

Listing Two

```
                {       cp = _submit+1;
                        for (i = 0; i < 127; ++i)
                        {       _bputchar(*cp);
                                if (*cp)
                                        ++cp;
                        }
                        if (_bputchar(*cp) == EOF)
                        {
error:                          fclose(fp);
                                return NULL;    /* overflow */
                        }
                } while (_submit = *_submit);

                /* signal CCP to select a: and user 0 */
                cp = DSKBYTE;
                *cp = 0;
        }

        /* exit this program and perform warm start */
        exit();
}
```

End Listing Two

Listing Three

```
#include <stdio.h>
extern  strcpy(), strcat(), submit(), puts();

main(argc, argv)
        int argc;
        int *argv;
{       int i;
        char buf[128];

        switch (argc) {
        default:
                strcpy(buf, "b:x ");
                for (i=2; i<argc; ++i)
                {       strcpy(buf+4, argv[i]);
                        submit(buf);
                }
        case 2:
                puts(argv[1]);
        case 1:
                ;
        }
        if (submit(NULL) == NULL)
                puts("oops");
}
```

End Listing Three

A New Library for Small-C [Part II]

In December 1982 and January 1983 we presented version 2.0 of the Small-C compiler by James Hendrix. Last month Hendrix and Ernest Payne presented their enhanced library for the compiler, as well as code which upgrades the compiler to version 2.1. This month completes the listing of the library. –Ed.

Small-C Library

Listing Two

(Continued from May issue page 81)

File: FSCANF.C

```
#define NOCCARGC  /* no argument count passing */
/*
** Yes, that is correct.  Although these functions use an
** argument count, they do not call functions which need one.
*/
#include stdio.h
/*
** fscanf(fd, ctlstring, arg, arg, ...) - Formatted read.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
fscanf(argc) int argc; {
  int *nxtarg;
  nxtarg = CCARGC() + &argc;
  return (_scan(*(--nxtarg), --nxtarg));
  }

/*
** scanf(ctlstring, arg, arg, ...) - Formatted read.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
scanf(argc) int argc; {
  return (_scan(stdin, CCARGC() + &argc - 1));
  }

/*
** _scan(fd, ctlstring, arg, arg, ...) - Formatted read.
** Called by fscanf() and scanf().
*/
_scan(fd,nxtarg) int fd, *nxtarg; {
  char *carg, *ctl, *unsigned;
  int  *narg, wast, ac, width, ch, cnv, base, ovfl, sign;
  ac = 0;
  ctl = *nxtarg--;
  while(*ctl) {
    if(isspace(*ctl)) {++ctl; continue;}
    if(*ctl++ != '%') continue;
    if(*ctl == '*') {narg = carg = &wast; ++ctl;}
    else            narg = carg = *nxtarg--;
    ctl += utoi(ctl, &width);
    if(!width) width = 32767;
    if(!(cnv = *ctl++)) break;
    while(isspace(ch = fgetc(fd))) ;
    if(ch == EOF) {if(ac) break; else return(EOF);}
    ungetc(ch,fd);
    switch(cnv) {
      case 'c':
        *carg = fgetc(fd);
        break;
      case 's':
        while(width--) {
          if((*carg = fgetc(fd)) == EOF) break;
          if(isspace(*carg)) break;
          if(carg != &wast) ++carg;
          }
        *carg = 0;
        break;
      default:
        switch(cnv) {
          case 'b': base =  2; sign = 1; ovfl = 32767; break;
          case 'd': base = 10; sign = 0; ovfl =  3276; break;
          case 'o': base =  8; sign = 1; ovfl =  8191; break;
          case 'u': base = 10; sign = 1; ovfl =  6553; break;
          case 'x': base = 16; sign = 1; ovfl =  4095; break;
          default:  return (ac);
          }
        *narg = unsigned = 0;
        while(width-- && !isspace(ch=fgetc(fd)) && ch!=EOF) {
          if(!sign)
            if(ch == '-') {sign = -1; continue;}
            else sign = 1;
          if(ch < '0') return (ac);
          if(ch >= 'a')      ch -= 87;
          else if(ch >= 'A') ch -= 55;
          else               ch -= '0';
          if(ch >= base || unsigned > ovfl) return (ac);
          unsigned = unsigned * base + ch;
          }
        *narg = sign * unsigned;
      }
    ++ac;
    }
```

(Continued on next page)

by James Hendrix and Ernest Payne

Copyright © 1984 L. E. Payne and J. E. Hendrix.

James Hendrix, Box 8378, University, MS 38677-8378.
Ernest Payne, 1331 W. Whispering Hills Dr., Tucson, AZ 85704.

Unix is a trademark of AT&T Bell Labs. MACRO-80 is a trademark of Microsoft Inc.

Small-C Library (Listing Continued)

Listing Two

```
86   return (ac);
87   }
88

   File: FWRITE.C

 1 #define NOCCARGC  /* no argument count passing */
 2 #include clib.def
 3 extern int _status[];
 4 /*
 5 ** Item-stream write to fd.
 6 ** Entry: buf = address of source buffer
 7 **          sz = size of items in bytes
 8 **           n = number of items to write
 9 **          fd = file descriptor
10 ** Returns a count of the items actually written or
11 ** zero if an error occurred.
12 ** May use ferror(), as always, to detect errors.
13 */
14 fwrite(buf, sz, n, fd) char *buf; int sz, n, fd; {
15   int cnt;
16   if((cnt = write(fd, buf, n*sz)) == -1) return (0);
17   return (cnt);
18   }
19
20 /*
21 ** Binary-stream write to fd.
22 ** Entry:  fd = file descriptor
23 **        buf = address of source buffer
24 **          n = number of bytes to write
25 ** Returns a count of the bytes actually written or
26 ** -1 if an error occurred.
27 ** May use ferror(), as always, to detect errors.
28 */
29 write(fd, buf, n) int fd, n; char *buf; {
30   char *cnt;  /* fake unsigned */
31   cnt = 0;
32   while(n--) {
33     _write(*buf++, fd);
34     if(_status[fd] & ERRBIT) return (-1);
35     ++cnt;
36     }
37   return (cnt);
38   }

   File: GETARG.C

 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 /*
 4 ** Get command line argument.
 5 ** Entry: n    = Number of the argument.
 6 **        s    = Destination string pointer.
 7 **        size = Size of destination string.
 8 **        argc = Argument count from main().
 9 **        argv = Argument vector(s) from main().
10 ** Returns number of characters moved on success,
11 ** else EOF.
12 */
13 getarg(n,s,size,argc,argv)
14   int n; char *s; int size, argc, argv[]; {
15   char *str;
16   int i;
17   if(n < 0 | n >= argc) {
18     *s = NULL;
19     return EOF;
20     }
21   i = 0;
22   str=argv[n];
23   while(i<size) {
24     if((s[i]=str[i])==NULL) break;
25     ++i;
26     }
27   s[i]=NULL;
28   return i;
29   }

   File: GETCHAR.C

 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 /*
 4 ** Get next character from standard input.
 5 */
 6 getchar() {
 7   return (fgetc(stdin));
 8   }

   File: ISALNUM.C

 1 /*
 2 ** return 'true' if c is alphanumeric
 3 */
 4 isalnum(c) int c; {
 5   return ((c<='z' && c>='a') ||
 6           (c<='Z' && c>='A') ||
 7           (c<='9' && c>='0'));
 8   }

   File: ISALPHA.C

 1 /*
 2 ** return 'true' if c is alphabetic
 3 */
 4 isalpha(c) int c; {
 5   return ((c<='z' && c>='a') || (c<='Z' && c>='A'));
 6   }

   File: ISASCII.C

 1 /*
 2 ** return 'true' if c is an ASCII character (0-127)
 3 */
```

```
4 isascii(c) char *c; {
5   /* c is a simulated unsigned integer */
6   return (c <= 127);
7   }
```

File: ISATTY.C

```
1 extern int _device[];
2 /*
3 ** Return "true" if fd is a device, else "false"
4 */
5 isatty(fd) int fd; {
6   return (_device[fd]);
7   }
```

File: ISCNTRL.C

```
1 /*
2 ** return 'true' if c is a control character
3 ** (0-31 or 127)
4 */
5 iscntrl(c) char *c; {
6   /* c is a simulated unsigned integer */
7   return ((c <= 31) || (c == 127));
8   }
```

File: ISCONS.C

```
1 #include stdio.h
2 #include clib.def
3 extern int _device[];
4 /*
5 ** Determine if fd is the console.
6 */
7 iscons(fd) int fd; {
8   return (_device[fd] == CPMCON);
9   }
```

File: ISDIGIT.C

```
1 /*
2 ** return 'true' if c is a decimal digit
3 */
4 isdigit(c) int c; {
5   return (c<='9' && c>='0');
6   }
```

File: ISGRAPH.C

```
1 /*
2 ** return 'true' if c is a graphic character
3 ** (33-126)
4 */
5 isgraph(c) int c; {
6   return (c)=33 && c<=126);
7   }
```

File: ISLOWER.C

```
1 /*
2 ** return 'true' if c is lower-case alphabetic
3 */
4 islower(c) int c; {
5   return (c<='z' && c>='a');
6   }
```

File: ISPRINT.C

```
1 /*
2 ** return 'true' if c is a printable character
3 ** (32-126)
4 */
5 isprint(c) int c; {
6   return (c)=32 && c<=126);
7   }
```

File: ISPUNCT.C

```
1 #define NOCCARGC  /* no argument count passing */
2 /*
3 ** return 'true' if c is a punctuation character
4 ** (all but control and alphanumeric)
5 */
6 ispunct(c) int c; {
7   return (!isalnum(c) && !iscntrl(c));
8   }
```

File: ISSPACE.C

```
1 /*
2 ** return 'true' if c is a white-space character
3 */
4 isspace(c) int c; {
5   /* first check gives quick exit in most cases */
6   return(c<=' ' && (c==' ' || (c<=13 && c>=9)));
7   }
8
```

File: ISUPPER.C

```
1 /*
2 ** return 'true' if c is upper-case alphabetic
3 */
4 isupper(c) int c; {
5   return (c<='Z' && c>='A');
6   }
```

File: ISXDIGIT.C

```
1 /*
2 ** return 'true' if c is a hexadecimal digit
3 ** (0-9, A-F, or a-f)
```

(Continued on page 60)

Small-C Library (Listing Continued)

Listing Two

```
4 */
5 isxdigit(c) int c; {
6   return ((c<='f' && c>='a') ||
7           (c<='F' && c>='A') ||
8           (c<='9' && c>='0'));
9   }
```

File: ITOA.C

```
1 #define NOCCARGC  /* no argument count passing */
2 /*
3 ** itoa(n,s) - Convert n to characters in s
4 */
5 itoa(n, s) char *s; int n; {
6   int sign;
7   char *ptr;
8   ptr = s;
9   if ((sign = n) < 0) /* record sign */
10    n = -n;     /* make n positive */
11  do {  /* generate digits in reverse order */
12    *ptr++ = n % 10 + '0';  /* get next digit */
13    } while ((n = n / 10) > 0);        /* delete it */
14  if (sign < 0) *ptr++ = '-';
15  *ptr = '\0';
16  reverse(s);
17  }
```

File: ITOAB.C

```
1 #define NOCCARGC  /* no argument count passing */
2 /*
3 ** itoab(n,s,b) - Convert "unsigned" n to characters in s
  using base b.
4 **               NOTE: This is a non-standard function.
5 */
6 itoab(n, s, b) int n; char *s; int b; {
7   char *ptr;
8   int lowbit;
9   ptr = s;
10  b >>= 1;
11  do {
12    lowbit = n & 1;
13    n = (n >> 1) & 32767;
14    *ptr = ((n % b) << 1) + lowbit;
15    if(*ptr < 10) *ptr += '0'; else *ptr += 55;
16    ++ptr;
17    } while(n /= b);
18  *ptr = 0;
19  reverse (s);
20  }
21
```

File: ITOD.C

```
1 #include stdio.h
2 /*
3 ** itod -- convert nbr to signed decimal string of width sz
4 **         right adjusted, blank filled; returns str
5 **
6 **        if sz > 0 terminate with null byte
7 **        if sz = 0 find end of string
8 **        if sz < 0 use last byte for data
9 */
10 itod(nbr, str, sz)  int nbr;  char str[];  int sz;  {
11   char sgn;
12   if(nbr<0) {nbr = -nbr; sgn='-';}
13   else sgn=' ';
14   if(sz>0) str[--sz]=NULL;
15   else if(sz<0) sz = -sz;
16   else while(str[sz]!=NULL) ++sz;
17   while(sz) {
18     str[--sz]=(nbr%10+'0');
19     if((nbr=nbr/10)==0) break;
20     }
21   if(sz) str[--sz]=sgn;
22   while(sz>0) str[--sz]=' ';
23   return str;
24   }
```

File: ITOO.C

```
1 /*
2 ** itoo -- converts nbr to octal string of length sz
3 **         right adjusted and blank filled, returns str
4 **
5 **        if sz > 0 terminate with null byte
6 **        if sz = 0 find end of string
7 **        if sz < 0 use last byte for data
8 */
9 itoo(nbr, str, sz)  int nbr;  char str[];  int sz;  {
10   int digit;
11   if(sz>0) str[--sz]=0;
12   else if(sz<0) sz = -sz;
13   else while(str[sz]!=0) ++sz;
14   while(sz) {
15     digit=nbr&7; nbr=(nbr>>3)&8191;
16     str[--sz]=digit+48;
17     if(nbr==0) break;
18     }
19   while(sz) str[--sz]=' ';
20   return str;
21   }
```

File: ITOU.C

```
1 #include stdio.h
2 /*
3 ** itou -- convert nbr to unsigned decimal string of width sz
```

```
 4 **         right adjusted, blank filled; returns str
 5 **
 6 **        if sz > 0 terminate with null byte
 7 **        if sz = 0 find end of string
 8 **        if sz < 0 use last byte for data
 9 */
10 itou(nbr, str, sz)  int nbr;  char str[];  int sz;  {
11   int lowbit;
12   if(sz>0) str[--sz]=NULL;
13   else if(sz<0) sz = -sz;
14   else while(str[sz]!=NULL) ++sz;
15   while(sz) {
16     lowbit=nbr&1;
17     nbr=(nbr>>1)&32767;  /* divide by 2 */
18     str[--sz]=((nbr%5)<<(1)+lowbit+'0';
19     if((nbr=nbr/5)==0) break;
20     }
21   while(sz) str[--sz]=' ';
22   return str;
23   }
```

File: ITOX.C

```
 1 /*
 2 ** itox -- converts nbr to hex string of length sz
 3 **         right adjusted and blank filled, returns str
 4 **
 5 **        if sz > 0 terminate with null byte
 6 **        if sz = 0 find end of string
 7 **        if sz < 0 use last byte for data
 8 */
 9 itox(nbr, str, sz)  int nbr;  char str[];  int sz;  {
10   int digit, offset;
11   if(sz>0) str[--sz]=0;
12   else if(sz<0) sz = -sz;
13   else while(str[sz]!=0) ++sz;
14   while(sz) {
15     digit=nbr&15; nbr=(nbr>>4)&4095;
16     if(digit<10) offset=48; else offset=55;
17     str[--sz]=digit+offset;
18     if(nbr==0) break;
19     }
20   while(sz) str[--sz]=' ';
21   return str;
22   }
```

File: LEFT.C

```
1 /*
2 ** left -- left adjust and null terminate a string
3 */
4 left(str) char *str; {
5   char *str2;
6   str2=str;
7   while(*str2==' ') ++str2;
8   while(*str++ = *str2++);
9   }
```

File: LEXCMP.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 /*
 3 ** lexcmp(s, t) - Return a number <0, 0, or>0
 4 **                as s is <, =, or > t.
 5 */
 6 lexcmp(s, t) char *s, *t; {
 7   while(*s == *t) {
 8     if(*s == 0) return (0);
 9     ++s; ++t;
10     }
11   return (lexorder(*s, *t));
12   }
13
14 /*
15 ** lexorder(c1, c2)
16 **
17 ** Return a negative, zero, or positive number if
18 ** c1 is less than, equal to, or greater than c2,
19 ** based on a lexicographical (dictionary order)
20 ** colating sequence.
21 **
22 */
23 char _lex[128] = {
24       /**** NUL - / ****/
25       000,001,002,003,004,005,006,007,008,009,
26       010,011,012,013,014,015,016,017,018,019,
27       020,021,022,023,024,025,026,027,028,029,
28       030,031,032,033,034,035,036,037,038,039,
29       040,041,042,043,044,045,046,047,
30       /**** 0-9 ****/
31       065,066,067,068,069,070,071,072,073,074,
32       /**** : ; < = > ? @ ****/
33       048,049,050,051,052,053,054,
34       /**** A-Z ****/
35       075,076,077,078,079,080,081,082,083,084,085,086,087,
36       088,089,090,091,092,093,094,095,096,097,098,099,100,
37       /**** [ \ ] ^ _ ` ****/
38       055,056,057,058,059,060,
39       /**** a-z ****/
40       075,076,077,078,079,080,081,082,083,084,085,086,087,
41       088,089,090,091,092,093,094,095,096,097,098,099,100,
42       /**** { | } ~ ****/
43       061,062,063,064,
44       /**** DEL ****/
45       101
46       };
47
48 lexorder(c1, c2) char c1, c2; {
49   return(_lex[c1] - _lex[c2]);
50   }
```

File: MALLOC.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 /*
 4 ** Memory allocation of size bytes.
 5 ** size  = Size of the block in bytes.
 6 ** Returns the address of the allocated block,
 7 ** else NULL for failure.
```

(Continued on next page)

Small-C Library (Listing Continued)

Listing Two

```
 8 */
 9 malloc(size) char *size; {
10   return (_alloc(size, NO));
11   }
```

File: OTOI.C

```
 1 #include stdio.h
 2 /*
 3 ** otoi -- convert unsigned octal string to integer nbr
 4 **          returns field size, else ERR on error
 5 */
 6 otoi(octstr, nbr)  char *octstr;  int *nbr;  {
 7   int d,t; d=0;
 8   *nbr=0;
 9   while((*octstr>='0')&(*octstr<='7')) {
10     t=*nbr;
11     t=(t<<3) + (*octstr++ - '0');
12     if ((t>=0)&(*nbr<0)) return ERR;
13     d++; *nbr=t;
14     }
15   return d;
16   }
```

File: PAD.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 /*
 3 ** Place n occurrences of ch at dest.
 4 */
 5 pad(dest, ch, n) char *dest, *n; int ch; {
 6   /* n is a fake unsigned integer */
 7   while(n--) *dest++ = ch;
 8   }
```

File: POLL.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 #include clib.def
 4 /*
 5 ** Poll for console input or interruption
 6 */
 7 poll(pause) int pause; {
 8   int i;
 9   i = _bdos(DCONIO, 255);
10   if(pause) {
11     if(i == PAUSE) {
12       while(!(i = _bdos(DCONIO, 255))) ;
13       if(i == ABORT) exit(0);
14       return (0);
15       }
16     if(i == ABORT) exit(0);
17     }
18   return (i);
19   }
```

File: PUTCHAR.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 /*
 4 ** Write character to standard output.
 5 */
 6 putchar(ch) int ch; {
 7   return (fputc(ch, stdout));
 8   }
```

File: PUTS.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 /*
 4 ** Write string to standard output.
 5 */
 6 puts(string) char *string; {
 7   fputs(string, stdout);
 8   fputc('\n', stdout);
 9   }
```

File: RENAME.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 #include clib.def
 4 /*
 5 ** Rename a file.
 6 **  from = address of old filename.
 7 **    to = address of new filename.
 8 **  Returns NULL on success, else ERR.
 9 */
10 rename(from, to) char *from, *to; {
11   char fcb[FCBSIZE];
12   pad(fcb, NULL, FCBSIZE);
13   if(!_newfcb(to, fcb) || _bdos(OPNFIL, fcb) != 255) {
14     _bdos(CLOFIL, fcb);
15     return (ERR);
16     }
17   if(_newfcb(from, fcb) &&
18      _newfcb(to, fcb+NAMEOFF2) &&
19      _bdos(RENAME, fcb) != 255)
20     return (NULL);
21   return (ERR);
22   }
```

File: REVERSE.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 /*
 3 ** reverse string in place
 4 */
 5 reverse(s) char *s; {
```

```
 6   char *j;
 7   int c;
 8   j = s + strlen(s) - 1;
 9   while(s < j) {
10     c = *s;
11     *s++ = *j;
12     *j-- = c;
13     }
14   }
15
```

File: REWIND.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 /*
 3 ** Rewind file to beginning.
 4 */
 5 rewind(fd) int fd; {
 6   return(cseek(fd, 0, 0));
 7   }
```

File: SIGN.C

```
 1 /*
 2 ** sign -- return -1, 0, +1 depending on the sign of nbr
 3 */
 4 sign(nbr)  int nbr;  {
 5   if(nbr>0) return 1;
 6   if(nbr==0) return 0;
 7   return -1;
 8   }
```

File: STRCAT.C

```
 1 /*
 2 ** concatenate t to end of s
 3 ** s must be large enough
 4 */
 5 strcat(s, t) char *s, *t; {
 6   char *d;
 7   d = s;
 8   --s;
 9   while (*++s) ;
10   while (*s++ = *t++) ;
11   return(d);
12   }
```

File: STRCHR.C

```
 1 /*
 2 ** return pointer to 1st occurrence of c in str, else 0
 3 */
 4 strchr(str, c) char *str, c; {
 5   while(*str) {
 6     if(*str == c) return (str);
 7     ++str;
 8     }
 9   return (0);
10   }
```

File: STRCMP.C

```
 1 /*
 2 ** return <0,   0,  >0 a_ording to
 3 **       s<t, s=t, s>t
 4 */
 5 strcmp(s, t) char *s, *t; {
 6   while(*s == *t) {
 7     if(*s == 0) return (0);
 8     ++s; ++t;
 9     }
10   return (*s - *t);
11   }
12
```

File: STRLEN.C

```
 1 /*
 2 ** return length of s
 3 */
 4 strlen(s) char *s; {
 5   char *t;
 6   t = s - 1;
 7   while (*++t) ;
 8   return (t - s);
 9   }
```

File: STRCPY.C

```
 1 /*
 2 ** copy t to s
 3 */
 4 strcpy(s, t) char *s, *t; {
 5   char *d;
 6   d = s;
 7   while (*s++ = *t++) ;
 8   return(d);
 9   }
```

File: STRNCAT.C

```
 1 /*
 2 ** concatenate n bytes max from t to end of s
 3 ** s must be large enough
 4 */
 5 strncat(s, t, n) char *s, *t; int n; {
 6   char *d;
 7   d = s;
```

(Continued on page 68)

Small-C Library (Listing Continued)

Listing Two

```
 8   --s;
 9   while(*++s) ;
10   while(n--) {
11     if(*s++ = *t++) continue;
12     return(d);
13     }
14   *s = 0;
15   return(d);
16   }
```

File: STRNCMP.C

```
 1 /*
 2 ** strncmp(s,t,n) - Compares two strings for at most n
 3 **                  characters and returns an integer
 4 **                  >0, =0, or <0 as s is >t, =t, or <t.
 5 */
 6 strncmp(s, t, n) char *s, *t; int n; {
 7   while(n && *s==*t) {
 8     if (*s == 0) return (0);
 9     ++s; ++t; --n;
10     }
11   if(n) return (*s - *t);
12   return (0);
13   }
```

File: STRNCPY.C

```
 1 /*
 2 ** copy n characters from sour to dest (null padding)
 3 */
 4 strncpy(dest, sour, n) char *dest, *sour; int n; {
 5   char *d;
 6   d = dest;
 7   while(n-- > 0) {
 8     if(*d++ = *sour++) continue;
 9     while(n-- > 0) *d++ = 0;
10     }
11   *d = 0;
12   return (dest);
13   }
```

File: STRRCHR.C

```
 1 /*
 2 ** strrchr(s,c) - Search s for rightmost occurrance of c.
 3 ** s      = Pointer to string to be searched.
 4 ** c      = Character to search for.
 5 ** Returns pointer to rightmost c or NULL.
 6 */
 7 strrchr(s, c) char *s, c; {
 8   char *ptr;
 9   ptr = 0;
10   while(*s) {
11     if(*s==c) ptr = s;
12     ++s;
13     }
14   return (ptr);
15   }
```

File: TOASCII.C

```
 1 /*
 2 ** return ASCII equivalent of c
 3 */
 4 toascii(c) int c; {
 5   return (c);
 6   }
```

File: TOLOWER.C

```
 1 /*
 2 ** return lower-case of c if upper-case, else c
 3 */
 4 tolower(c) int c; {
 5   if(c<='Z' && c>='A') return (c+32);
 6   return (c);
 7   }
```

File: TOUPPER.C

```
 1 /*
 2 ** return upper-case of c if it is lower-case, else c
 3 */
 4 toupper(c) int c; {
 5   if(c<='z' && c>='a') return (c-32);
 6   return (c);
 7   }
```

File: UNGETC.C

```
 1 #define NOCCARGC  /* no argument count passing */
 2 #include stdio.h
 3 extern _nextc[];
 4 /*
 5 ** Put c back into file fd.
 6 ** Entry:  c = character to put back
 7 **         fd = file descriptor
 8 ** Returns c if successful, else EOF.
 9 */
10 ungetc(c, fd) int c, fd; {
```

```
11   if(!_mode(fd) || _nextc[fd]!=EOF || c==EOF) return (EOF);
12   return (_nextc[fd] = c);
13   }
```

File: UNLINK.C

```
 1 #define NOCCARGC  /* no arg count passing */
 2 #include stdio.h
 3 #include clib.def
 4 /*
 5 ** Unlink (delete) the named file.
 6 ** Entry: fn = Null-terminated CP/M file name.
 7 **             May be prefixed by letter of drive.
 8 ** Returns NULL on success, else ERR.
 9 */
10 unlink(fn) char *fn; {
11   char fcb[FCBSIZE];
12   pad(fcb, NULL, FCBSIZE);
13   if(_newfcb(fn, fcb) && _bdos(DELFIL, fcb) != 255)
14     return (NULL);
15   return (ERR);
16   }
17 #asm
18 delete  equ    unlink
19         entry  delete
20 #endasm
```

File: UTOI.C

```
 1 #include stdio.h
 2 /*
 3 ** utoi -- convert unsigned decimal string to integer nbr
 4 **         returns field size, else ERR on error
 5 */
 6 utoi(decstr, nbr)  char *decstr;  int *nbr;  {
 7   int d,t; d=0;
 8   *nbr=0;
 9   while((*decstr>='0')&(*decstr<='9')) {
10     t=*nbr;t=(10*t) + (*decstr++ - '0');
11     if ((t>=0)&(*nbr<0)) return ERR;
12     d++; *nbr=t;
13     }
14   return d;
15   }
```

File: XTOI.C

```
 1 #include stdio.h
 2 /*
 3 ** xtoi -- convert hex string to integer nbr
 4 **         returns field size, else ERR on error
 5 */
 6 xtoi(hexstr, nbr)  char *hexstr;  int *nbr;  {
 7   int d,t; d=0;
 8   *nbr=0;
 9   while(1)
10     {
11     if((*hexstr>='0')&(*hexstr<='9')) t=48;
12     else if((*hexstr>='A')&(*hexstr<='F')) t=55;
13     else if((*hexstr>='a')&(*hexstr<='f')) t=87;
14     else break;
15     if(d<4) ++d; else return ERR;
16     *nbr=*nbr<<4;
17     *nbr=*nbr+(*hexstr++)-t;
18     }
19   return d;
20   }
```

End Listings

Comments on "Sixth Generation Computers"

The dialogue between Michael Doherty and Richard Grigonis stretches back over the past year and a half. Having begun things in December 1982 with a brief discussion of fifth generation computers, Grigonis held forth last month on what to expect in the sixth generation. Here are Doherty's thoughts in reply. – Ed.

Most readers may recall my previous confrontations with Richard Grigonis in the pages of *Dr. Dobb's Journal* and how ridiculous he made me look with his article "And Still More Fifth Generation Computers" in August 1983. Naturally, it was with a feeling of great foreboding that I recently opened an envelope bearing the corporate logo of the Children's Television Workshop, as it could mean only one thing: Grigonis was sending me an advance copy of his latest article ("Sixth Generation Computers") and was daring me to do something about it!

Upon reading the article, however, I was relieved to discover that Grigonis's vast intellect had finally confounded itself and that I may yet emerge victorious from a final confrontation. Grigonis doesn't make it any easier on himself when he writes that the favorable response to his last article has encouraged him to write yet another opus, "taking speculation about future computers to the limits that can be conceived of by the human mind." His modest claims remind me of the tale of Erasmus from those strange post-Biblical writings known as the Apocrypha. The story goes that Erasmus, most learned man on Earth, wishes to demonstrate that, like Jesus, he can ascend even unto heaven, and so he begins to float up into the sky. Saint Peter, appalled at this flagrant display, prays to God that something be done, whereupon Erasmus falls to Earth and breaks his arms and legs. Saint Peter then breathes a sigh of relief and sternly notes to the reader that "here was a man who could fly like a bird, and now he cannot even crawl like a worm."

by Michael Doherty

Michael J. Doherty, 334 South Maple Avenue, Glen Rock, New Jersey 07452.

Grigonis's predictions this time around include reducing the execution time of artificial intelligence (AI) programs by building processors having signals that can travel faster than light, which incidentally allows the computer to answer questions even before the programs asking the questions are run! His other prediction is that we will be able to communicate with such "superluminal" sixth generation computers through a technique whereby the computer reads the user's mind! These ideas definitely beg closer scrutiny, so let us continue.

First of all, most physicists at one time would have agreed with Grigonis's statement that "there is nothing in relativity theory, quantum mechanics, electrodynamics, or mechanics that says that time must move in one direction," but a little item from quantum mechanics may one day prove otherwise: namely, the controversy over the conservation of charge conjugation, parity reflection, and time reversal, the so-called "CPT theorems" initially stated by Wolfgang Pauli in 1942 and proved by the Swiss physicist Res Jost in 1958. The strong possibility exists that certain subatomic processes can move only forward in time. On the other hand, violations of the conservation laws may be possible, and we therefore might have to throw out most of physics, but this is unlikely. The whole question is still up in the air and is certainly not as open-and-shut as Grigonis states.

Physicist Bryce S. DeWitt, however, writing in a recent issue of *Scientific American*, discusses the idea of "quantum gravity" and how a subatomic event that is quick enough, or of a high enough energy, could reveal the quantum "grainy" structure of space-time, thereby blurring the distinction between the past and future.[1] In more scientific terms, if gravitation (and, hence, space-time) is quantum in nature, then the shape of the light cone defining the regions of space-time that are accessible from a given point in space and moment in time would act as if it were "fuzzy." Therefore, the ordinary accepted phenomenon of two points in space-time communicating with each other by means of signals moving equal to or less than the speed of light can be given only a probabilistic certainty. At tiny Planckian dimensions (1.61×10^{-33} centimeter), at Planckian time units (5.36×10^{-44} second), and at high enough energies, the probability of a signal traveling faster than the speed of light between two points could be very high indeed or at least high enough to make one of Grigonis's superluminal processors possible.

But to test Grigonis's theory, the energies required to explore this sub-subatomic realm would be immense. As DeWitt writes in his article: "To probe these scales of distance and time experimentally, using instruments built with present technology, one would need a particle accelerator the size of the galaxy!" I somehow doubt that even IBM would want to build one of Grigonis's proposed sixth generation computers if it entailed having to construct superluminal processor switches the size of the Milky Way!

Grigonis tries to get around this with all sorts of ingenious arguments that hover at the most distant horizon of theoretical physics. I would like to comment on them but I don't feel that I'm qualified. Indeed, I can think of only a handful of physicists in the world who are capable of seriously evaluating some of the things he suggests, which makes me wonder how he thought of them himself!

As for the "mind-reading" aspects of his proposed computer, I also find these difficult to disprove, but only because the physics is once again beyond my level of training. In fact, I have a sneaking suspicion that Grigonis may even be right! Richard Feynman and Steven Weinberg, where are you when we need you?

Moreover, Grigonis has reduced all of AI to a matter of searching through a search space that consists of the logical relationships of every known fact or assertion. Instead of resorting to forms of constrained, heuristic search, Grigonis astonishes us with his suggestion to stay with the "brute force" method and simply build computers with processors having signals moving faster than the speed of light.

Grigonis warns us that such superluminal computers would answer our questions long before we ever ask them. He proposes to solve this problem by placing every query in a buffer where a conventional computer and AI program "would analyze the parameters of the question and estimate the appropriate amount of time to wait before sending the question to the superluminal processor." Grigonis goes on to assure us that, "if the statement was accurate, the delay period of submitting the question to the superluminal processor would be just a little longer than the negative time required for an answer . . . ." The answer

thus would appear immediately after you enter the question instead of sometime in the past.

The problem with making a "rough estimate" of the execution time of any program is that, because of such things as Gödel's Proof and Turing's Halting Problem, such an estimate can only be rough, never precise: there is no way to determine whether a Turing machine (digital computer) is ever going to stop running any particular program. As Gödel discovered, there are "undecidable propositions" in logic; certain problems take an infinite amount of time to solve because they are unsolvable, and no logical way exists to determine if the computer is ever going to stop. All of this means that, if one of Grigonis's sixth generation computers were studying a really difficult, "interesting" problem, then the estimate of how long to hold the query in the buffer would certainly be off by at least an order of magnitude, and the answer could appear unpredictably in the past, present, or future!

Grigonis ignores another plan that does not require a superluminal processor, but here too the argument breaks down. I am referring to the constant trade-off of processing time versus storage size. Instead of using simple assertions and a resolution problem solver, one could make the knowledge representation of a particular problem more powerful by storing large, complex frames or semantic nets on disk.

For example, Grigonis is correct when he says that, given an initial chess position, up to 10^{150} move sequences are possible, requiring eons of processing time to search through. However, let's imagine that we've already calculated every possible move in every possible chess game *once* and have stored them all on a gigantic disk drive in the form of a series of records or vectors. With such an extreme form of knowledge representation, the processing time becomes almost zero as the problem of making the next move is now reduced to looking up entries in a table; the program simply looks up the current configuration of chess pieces on its internalized "chart" and immediately sees the entry that tells it the best move to make next. Deduction as such would be unnecessary, since this or any other problem situation could in theory be made complete in terms of all relevant objects, properties, and relations.

Of course, we have traded one impossibility for another: instead of an infinite amount of processing time, we now must provide an infinite number of disk drives! Besides, one rarely has a complete description of any problem situation, and a representation formalism based on logic does make it possible to express generalizations economically in spite of this. Still, just because we can deduce the behavior (the various possible states) of a system from its description as a set of subsystems or even simple assertions does not imply that the system can be *explained* from any kind of modular, logical, representation/resolution, theorem-proving process. AI researchers had hoped to develop logical, problem-independent solution methods, but we now know from metamathematical grounds that no general, efficient, problem-independent solution technique exists.[2,3]

And what if the fundamental concepts themselves cannot be broken down into their component features with any utility? E. Rosch,[4,5] K. Nelson,[6] and D. S. Palermo,[7] just to name a few, have complained that the feature theory of semantic development in human language is wrong, that things are more complicated than just applying good old reductionism to the problem of meaning.

In feature theory, features are abstracted from a whole concept so that the concept's meaning can be known. But one must already know that whole concept in order to know what features to abstract from it. Also, as Nelson says, like Humpty Dumpty there is no way of putting the abstracted features together to form the original whole. As Palermo points out, "a list of features is not a concept."

Names or words referring to categories are acquired as a whole and not feature by feature. They are highly structured internally but paradoxically do not have sharp, distinct boundaries; therefore, things are classified not by their features but by their "distance" from an ideal prototype or "core exemplar," which sounds suspiciously like Plato's old philosophy of ideals and his doctrines of *form* and *anamnesis*. Rosch shows that categories are universal across languages because they are intrinsic biological structures given genetically to the brain from its evolutionary heritage.

All of this would explain why, like the conceptual dependency theory of semantic primitives in AI programs, the feature theory of human semantic acquisition ignores the existence of metaphors, even though it is almost impossible in practice to differentiate between metaphors and nonmetaphors. We can speak nonmetaphorically of "the long arm of the man," but we can also easily comprehend "the long arm of the crane," or "the long arm of the law." All words have some metaphoric power, the ability to overlay two different objects or events. As Marvin Minsky has written: "What something means to me depends on everything else I know . . . every meaning is built on other meanings, with no special place to start."[8]

Similarly, it also seems impossible to assign items to a finite number of categories and then state rules in terms of membership in those categories. John Ross, for instance, observed that certain verb structures could generally be used in place of a noun phrase with varying degrees of acceptability.[9] For example, the gerund "his going" can be used in the same positions as a normal noun phrase:

I regretted *his going*.

His going surprised me.

After *his going* we hired a new worker.

His going's main effect was to end the feud.

Ross thought that properties like "nouniness" could be placed along a scale he called a *squish*. His attempts to devise a rigorous formalism for squishes have not met with success, and AI researchers in the natural language processing field tend to sweep them under the rug.

So it seems that a computer program can have only a syntax – no real semantics. Of course these squishes, like the other feature/category problems, could be explained if the human brain were viewed not as a discrete machine like a Turing machine or a digital computer but as a series of parallel analog devices.

This brings us to some fundamental problems that Grigonis ignores. Since a universal Turing machine can simulate any calculating device or physical expression of any algorithm, and assuming that the mind arises out of the physical workings of the brain, it follows that the physical embodiment of a Turing machine (the digital computer) can simulate human cognition.

But what if the human brain is not a Turing machine? What if it is some kind of nondiscrete analog device? This would explain how it can comprehend continuous spectra of physical state changes without analyzing each discrete state. Or worse yet, what if the brain is a series of parallel analog processors?

John Searle exposed an even more fundamental misconception among AI researchers when he demonstrated that cognition is not solely a matter of formal symbol manipulation.[10] Most AI researchers think that "the mind is to the brain as the program is to the hardware," which implies that a Turing machine in the form of a program running on a digital computer can achieve cognition. This ignores the fact that, while even primitive AI programs can do things that people can do, they utilize only simple computations that have nothing to do with the kind of actual cognitive states experienced by humans.

After all, the strange fact about AI has always been that it's easier to simulate an expert than to simulate the general common sense of a five-year-old. Daniel Bobrow's STUDENT program could solve high school algebra word problems, and James Slagle's SAINT and Joel Moses's SIN programs could solve rather complex, abstract calculus problems. Amazingly, these programs relied on only about a hun-

dred rule-based "facts" to achieve their mathematical miracles, wheras Terry Winograd's SHRDLU program, designed to manipulate a simple world of children's building blocks, was over 80,000 words long and required many more rules than most of the "expert" systems!

As Searle says, "simulation is not duplication." A person with a photographic memory, after all, could memorize a program and "think it through," immediately realizing that no intentional states are involved, just a set of instructions having nothing to do with "understanding." As Searle writes: "No one supposes that computer simulations of a five-alarm fire will burn the neighborhood down or that a computer simulation of a rainstorm will leave us all drenched. Why on earth would anyone suppose that a computer simulation of understanding actually understood anything?" AI researchers are still under the spell of the Turing Imitation Game, with its strong smell of behaviorism and operationalism.

Ironically, the idea that cognition can be expressed in a set of logical transformations – a computer program – set apart from the hardware, be it computer or brain, implies a strange mind/body dualism that goes against the avowed materialistic stance of AI researchers. Suppose the biochemical reactions of the brain itself are responsible for cognition? As Hubert L. Dreyfus has pointed out, knowledge representation with formal symbols and transformations may be unnecessary in human cognition: " . . . these are usually nonformal representations, more like images, by means of which I explore what I *am*, not what I *know*. We thus appeal to *concrete* representations (images or memories) based on our own experience without having to make explicit the strict rules and their spelled out *ceteris paribus* conditions required by *abstract* symbolic representations."[11]

So although humans are sometimes annoyingly imperfect thinkers, at least they do think, unlike AI programs and the computers that run them. The fundamental workings of the universe since the Big Bang seem to legislate against building one of Grigonis's "ultraintelligent machines."

In closing, let me thank *Dr. Dobb's Journal* for allowing me to once again continue the "Grigonis-Doherty debate" within its pages. I can think of no other publication in this country with a readership capable of appreciating such esoterica. And, despite appearances to the contrary, I would also like to thank the Grand Master of Computerdom himself, Richard Grigonis, for writing the most brilliant and daring series of articles of the past 30 years, or at least since the legendary days of Alan Turing and Johnny von Neumann.

References

[1] DeWitt, Bryce S. Quantum gravity. *Scientific American*, Dec. 1983, Vol. 249, pp. 112-129.

[2] Davis, M. *Computability and Unsolvability*. New York: McGraw-Hill, 1958.

[3] Rabin, M. Theoretical impedience to artificial intelligence. *Proc. Conf. Inter. Fed. Info. Processing Societies*, 1975, Stockholm.

[4] Rosch, E. On the internal structure of perceptual and semantic categories. In T. E. Moore (ed.), *Cognitive Development and the Acquisition of Language*, New York: Academic Press, 1973, pp. 111-144.

[5] Rosch, E. Cognitive representation of semantic categories. *Journal of Experimental Psychology*, 1975, Vol. 104, pp. 192-233.

[6] Nelson, K. Concept, word, and sentence: Interrelations in acquisition and development. *Psychological Review*, 1974, Vol. 81, pp. 267-285.

[7] Palermo, D. S. Semantics and language acquisition: some theoretical considerations. Paper presented at the Psychology of Language Conference, University of Stirling, June 1976.

[8] Minsky, Marvin. Why people think computers can't. *Technology Review*, Nov./Dec. 1983, Vol. 86, pp. 65-81.

[9] Ross, John R. *Constraints on Variables in Syntax*. Ph.D. dissertation, MIT, 1967.

[10] Searle, John R. Minds, brains, and programs. In John Haugeland (ed.), *Mind Design*, Cambridge: The MIT Press, 1981, pp. 282-306.

[11] Dreyfus, Hubert L. From micro-worlds to knowledge representation: AI at an impasse. In John Haugeland (ed.), *Mind Design*, Cambridge: The MIT Press, 1981, pp. 161-204.

■■J

How many times have you said to yourself, "What this industry (still) needs is a good, low-cost Pascal compiler"? We've said the same thing ourselves, and we were fascinated to learn about Turbo Pascal. Not only low in price, it was advertized to be fast, take very little space, and include a resident editor. In addition, from everything we can determine, Borland ships promptly and provides good support. They have even disposed of their $100 commercial licensing fee.

Sound too good to be true? Well, the following review suggests that Turbo Pascal is what it claims. By the time you read this (lags in publishing being what they are) you will probably have seen ads in DDJ *for a version 2.0 that provides even more features, including automatic overlays, windows (at least for IBM PC and PC Jr.), and a dispose function. We will provide an update on that newer version in the next few months. In the meantime, Borland deserves watching. With similar packages for Modula-2 and C coming soon, their potential impact should not be underestimated. – Ed.*

Turbo Pascal, V.1.01

Company: Borland International, 4807 Scotts Valley Drive, Scotts Valley, CA 95066
Computer: CP/M-80, MS-DOS, or CP/M-86
Price: $49.95

Reviewed by David D. Clark

The advertisements say that "This is the Pascal compiler everybody's been waiting for." They may be right. Turbo Pascal, from Borland International, is an excellent product at an extraordinary price. Some people may be leery of a low-priced Pascal compiler after having had a bad experience with JRT Pascal. I myself had similar feelings when I saw the advertisements for Turbo Pascal. Since using the product, though, I have been pleasantly surprised. *Amazed* might be a better word.

When you open the package, you find a disk and a reference manual. The disk contains several programs and text files. TURBO.COM (or TURBO.CMD), TURBOMSG.OVER, and TURBO.OVR make up the compiler/editor, error messages, and a temporary program loader. Also present are an installation program and associated data file, a program-listing utility, an errata file, and the source files for a simple spreadsheet program called MicroCalc.

Turbo Pascal has many interesting and innovative features. It is very small, only about 28K. A full-screen editor is co-resident with the compiler. The compiler is exceptionally fast. Versions are available for both 8-bit and 16-bit systems. The user can develop highly standard, portable programs. The programs are reasonably fast. A floating-point format with eleven decimal digits allows business applications to handle dollars and cents calculations to $999,999,999.99. Access is provided to the underlying hardware and operating system facilities. In-line machine language can be generated, and the user can control compiler operation with a variety of directives.

The Editor

The built-in, full-screen editor is one of the major conveniences this package provides. It is co-resident with the compiler and, as such, allows a speed of interaction during program development that is unprecedented with a compiled language. It is possible to enter the editor part of the package, create the source text for a program, exit from the editor, compile the text in memory, and immediately return to the editor in the event of a compilation error, all without any disk access required. It's fast.

The first thing you will want to do after scanning the reference manual is install the editor for your computer. This is done by running the installation program TINST. There are two parts to the process: terminal installation and command installation. The first and most important is installing the correct terminal driver for your CRT. TINST uses a data file containing information about a number of common terminals. If yours is one of those already known to the system, you will just have to select it from a menu and the program will finish the installation. If your terminal is not one of those listed, or if you wish to change the default parameters for a listed terminal, you will be led through a series of questions that will set up the system to run with your equipment. I tried the installation procedure with three different terminals. The system had menu entries for two of them, a Zenith and a TeleVideo, and installed them without a hitch. I also installed the system on a North Star Advantage, which is not listed in the menu. The installation process was clear and quick – you just need to know the control codes for your particular terminal.

Turbo Pascal has several procedures that are extensions to standard Pascal and that allow control of the video screen of your terminal. You must complete the installation process correctly in order to guarantee that these procedures work as expected.

The default commands for the editor are a subset of those used by WordStar. Cursor movement, insertion, deletion, find, replace, and the block commands are almost identical with those of WordStar. Some additional commands are also present. The editor's differences from WordStar, such as its incorporation of automatic indentation and a single command to mark a word as a block, make the editor easier to use in a program-development environment than WordStar. If you know WordStar, you will be able to use the Turbo editor.

The installation program also allows you to change the characters used to invoke each command. If you don't like the default choices, it's easy to alter them. Just remember to write down your changes somewhere.

Running Turbo Pascal

The heart of the system consists of the program TURBO. While running, TURBO is a simple program-development environment. When you first start TURBO, it informs you of the terminal it has been installed for and asks if you wish to include error messages. If you respond with an affirmative, a text file containing the compiler error messages will be loaded. These contain the text that will be displayed in the event the compiler detects an error while translating a program. If you do not load the error messages, you will have about 1.5K additional free memory, but the compiler will just print out a number when it detects an error. After the error messages are loaded (or not), a menu is displayed on the screen. The available options are shown in Table 1 (page 75).

Speed

A rather large demonstration program is included as a group of source files on the distribution disk. It is a simple spreadsheet program called MicroCalc. The first thing I did after backing up the master disk (even before installing

the editor) was compile this program. That was when I got my first surprise. The 1261-line program was compiled to a 22K object file in almost exactly one minute. This is just about the fastest compilation speed I have ever seen on an 8-bit microcomputer.

When compilations are done in memory (selected with the Options menu), the process is even faster because there are no disk-write accesses to slow things down. It is a truly impressive thing to see. I ran a couple of benchmarks to see how fast the generated code is. The first was the (in)famous *Byte* sieve prime-number generator. The program compiled almost instantly in memory, too fast for me to get an accurate measurement by hand. It generated 301 bytes of code (not including the data space occupied by the large Boolean array). The program executed in 24 seconds. Although this is not the fastest execution speed, it isn't too shabby either. It is much faster than UCSD or JRT Pascal and several seconds faster than the C version of the program when compiled with Aztec C II or Eco-C on my system.

Another benchmark I tried was the floating-point program from *Dr. Dobb's* No. 83 and No. 89. Because Turbo Pascal does not include a standard function for calculating the arctangent, the program declared a function to calculate one. This program took 337 seconds to execute (again compilation was almost instantaneous) and had an error of 4.6E-3 from an exact answer of 2500. This is not too bad for software floating point on an 8-bit machine, especially when you consider that the floating-point format in Turbo Pascal uses a 40-bit mantissa (about 11 significant decimal digits), compared to 24 bits (about 7 significant decimal digits) in normal single-precision.

The Reference Manual

The reference manual is a paper-bound book of about 260 pages. The main part of the manual describes the features common to both the 8-bit and 16-bit versions of Turbo Pascal. The first two appendices, each about 30 pages, describe those parts of the system, specific to either the 8- or 16-bit version. The appendix describing the 16-bit version is further divided into sections covering the MS-DOS version and the CP/M-86 version. There are also a number of additional appendices that list the standard procedures and functions, operators, compiler directives (described below), differences from standard Pascal (also described below), compiler error messages, runtime error messages, I/O error messages, translation of error messages to foreign languages, detailed installation procedures, Turbo Pascal syntax, the ASCII character set, and a subject index.

Although the manual states that it is not intended to teach Pascal, it does cover the language features thoroughly with lots of examples. Unfortunately, there are some rough spots. There is a tendency to use *e.g.* with annoying frequency and often at an inappropriate place in a sentence. The same thing occasionally happes with *i.e.* A word processor appears to have been allowed free rein with decisions on where to hyphenate words. For example, "sc-reen" and "th-rough" are two of the more blatant blunders.

These errors aren't really too important. Every book of this size has some typographical or usage errors. Even reviews such as this one are not immune. *Real* confusion can occur when a program example is in error or does not agree with the surrounding text. For example, in the discussion of operations on text files, an example program declares a variable of **type** Text named "F." In the body of the program, however, all text-file operations are performed on an undeclared variable "FilVar."

If there is a weak point in the system, it is the reference manual. It isn't too bad, but it isn't anything great either. Mostly it's just sloppy. It needs one more thorough edit.

Deviations from Standard Pascal

The degree to which a particular implementation of a programming language adheres to the language standard affects the portability of programs developed with that implementation. Turbo Pascal uses the *Pascal User Manual and Report* by Kathleen Jensen and Niklaus Wirth, *not* the ISO standard, as the definition of standard Pascal. For most users,

Logged drive	Allows you to change the currently logged drive.
Work file	The work file is the file maintained in memory for alterations with the editor.
Main file	This command is used when the work file contains a file that is included from some other main program. If you later elect to compile a program, the work file will be saved and the main file will be read into memory for compilation.
Edit	Takes you to the editor to create or alter the work file.
Compile	Compile the work file or main file if different from the work file. A successful compilation can result in a "COM" (or "CMD") file, a "CHN" file, or a memory resident program, depending upon the selections made with the Options menu item.
Run	This command will run a memory resident or executable file. If the program has not been compiled, the compiler will be invoked automatically.
Save	Used to save an altered work file back to disk. Any previous version is renamed with the file extension "BAK" before the new version is saved.
Execute	Will run another program from TURBO. When the program terminates, control is transferred back to the TURBO program.
Directory	Allows viewing the contents of a disk directory. Disk specifiers and wild cards are allowed.
Quit	Exits the TURBO program. If the work file has been altered but not saved, you will be asked if you want to store it before leaving the program.
Options	This command varies slightly in the 8-bit and 16-bit versions. It allows you to direct the compiler to construct a program in memory or to disk. If a disk file is selected, it can be one of two types: an executable "COM" (or "CMD") file or a "CHN" file. A "CHN" file can be "chained" to or from another Turbo Pascal program. It does not contain the runtime library because one will be present from the calling program. Because no library is present, "CHN" files are about 8K smaller than executable files. It is also possible to specify the starting address of a program and the last memory location available to the program with the 8-bit version. The 16-bit version allows the user to specify a minimum code-segment size, a minimum data-segment size, and the minimum and maximum allowable dynamic memory. The options command also allows you to find where a runtime error occurred in a "COM" or "CHN" file.

Table 1

the difference between the two standards will be small, the largest difference being the inclusion of conformant arrays in the ISO standard. I'm not aware of a microcomputer-based Pascal that implements conformant arrays yet, anyway.

One of the appendices of the Turbo reference manual discusses deviations from the Jensen and Wirth standard. These fall into (basically) six areas and are mostly concessions to efficiency; that is, abandoning facilities that are difficult to implement. The deviations discussed do not include the many extensions available in the Turbo version. The deviations are:

1. The standard procedure **Dispose** is not available. Instead, you can manage heap space (used for dynamic variable allocation) by use of the standard procedures **Mark** and **Release**. The dynamic-variable allocation procedure **New** is also restricted; the Turbo version does not allow variant record specifications.
2. The standard I/O (input/output) procedures **Get** and **Put** are not implemented. Instead, the **Read, ReadIn, Write**, and **WriteIn** have been extended to handle all I/O.
3. The **Goto** statement is restricted in that it may transfer control only within the current block.
4. The standard procedure **Page** is not implemented.
5. Variable packing and unpacking are not under user control. Variables are packed whenever possible. Because of this, the reserved word **packed** has no effect and the standard procedures **Pack** and **Unpack** are not implemented.
6. A program may not pass procedures and functions as parameters to other subroutines.

The discussion of the **with** statement in the main body of the reference manual would also lead you to believe it is nonstandard. According to the manual, the user must dereference successive fields within records with commas instead of with periods as in standard Pascal. The number of records that the user can open using a **with** statement is also restricted, but the user can change that number by use of the "W" compiler directive. The version of the compiler I used was capable of handling the standard syntax as well. In fact, on all the tests I ran, using the standard form generated slightly less code than the technique presented in the reference manual.

Finally, the error messages are not the same as in the *Pascal User Manual and Report*. This is not really a standard, but most compilers based on the P4 portable compiler, developed by Professor Wirth's colleagues in Zurich, do use the same set of error messages.

I transferred several programs written for the UCSD version of Pascal to CP/M for use with Turbo Pascal. In most cases, the revisions needed to get the programs running under Turbo Pascal were minor, usually involving things that the UCSD system implements in a nonstandard manner, like opening files. I did have some difficulty in converting programs that made extensive use of the **Get, Put,** and **Dispose** standard procedures.

Extensions

Turbo Pascal provides a large number of extensions to standard Pascal. They usually support a simple means of doing something that is clumsy or impossible within the constraints of the standard language – for example, one provides direct access to the underlying hardware. These extensions may be convenient, but programs developed with them will be less portable than those that use only the facilities available in the standard.

The extensions available are numerous, so I will briefly summarize some of them. To paraphrase from the user's manual, they include:

1. Dynamic strings. Strings are almost indispensable for many programs. Most implementations of Pascal on microcomputers provide strings of some sort, so they are almost a standard. Turbo Pascal strings, and standard procedures and functions to manipulate them, closely resemble those available in other implementations such as JRT Pascal and UCSD Pascal.

The directives common to both versions are:

B Controls which physical devices are attached to the standard files **Input** and **Output**. When active (the default condition), the CON: device is used. When inactive, **Input** and **Output** are attached to the TRM: device. The CON: device provides buffered input with limited editing capabilities, while the TRM: device does not. This directive affects the entire program and may not be switched on and off within the program.

C When active, as it is by default, this option allows the user to interrupt program execution by typing a Control-C in response to a **Read** or **ReadIn** statement. You can use Control-S to halt console output temporarily.

I This directive has two functions. When followed by a file name, the contents of the file named will replace the comment in the compilation. The other use turns I/O error handling on (the default) or off. If an I/O error occurs while error checking is active, the program will be terminated. If checking is off, the user program is responsible for error detection by use of the standard function **IOError.**

R This directive determines whether runtime index-checking of arrays and strings is performed. Checks are turned off by default. Until a program is thoroughly debugged, checking should be turned on; otherwise, an index error can cause a program to crash – or worse, could cause undetected damage to data.

V This directive, which is normally on, determines how rigorous type-checking is when strings are passed as parameters. While active, strings passed as variable parameters must have been declared with the same maximum dynamic length as the formal parameter type. When inactive, the compiler will allow you to pass an actual parameter with a maximum length different from the formal parameter of a subroutine.

U If this directive is active, the user can interrupt the compiled program at any time by typing a Control-C. If inactive, as it is by default, Control-C will have no effect. Activating this option significantly decreases speed and slightly increases code size.

X This directive controls speed versus size optimization during calculation of array indices. When active (the default), optimizations are made for speed at the expense of code size. With the programs I tested, this option had an effect only when arrays with two or more dimensions were present.

The directives unique to the 8-bit implementation are:

A When active (the default state), the compiler will generate code for nonrecursive subroutines. When switched off, the code generated will allow recursive calls with a penalty in code size.

W The allowed nesting of **with** statements is controlled by the setting of this variable. It initially allows a nesting of two levels. As mentioned earlier, this directive has an effect only when you use the nonstandard method of opening nested records described in the manual. This directive has no effect if you use standard syntax in **with** statements.

Table 2.

2. Variable declarations specifying an absolute address to be occupied by the variable.
3. Bit and byte manipulation.
4. Direct access to memory, data ports, and operating-system facilities.
5. Relaxed restrictions on the ordering of **const, type,** and **var** declarations.
6. Ability to generate in-line machine code.
7. File inclusion from the main program.
8. Logical operations on integers, such as left and right shifts.
9. Program chaining with common variables.
10. Random-access data files.
11. Structured constants that allow a form of initialized variable declaration.
12. Type-conversion functions.
13. A large number of built-in routines to allow control of video terminals.

Extensions have also been made to the syntax of some statements. For example, the **case** statement allows an optional **else** part and subranges in case labels.

Compiler Directives

The user can control the operation of the compiler by use of several compiler directives included in programs as a special form of comment. There is a group of directives common to both the 8-bit and 16-bit versions of the system, as well as some present only on the 8-bit or 16-bit versions. These are shown in Table 2 (page 76).

There is only one directive specific to the 16-bit versions of Turbo Pascal. When the "K" directive is active, as it is initially, a check is made for adequate stack space before each subroutine call. When the directive is turned off, no such checks are performed.

Chaining and External Procedures

There are no true separate compilation facilities available with Turbo Pascal. Because compilation is so fast, however, it is almost as easy to develop files of commonly used routines and include them in the compilation of the main program as it would be with a separate-compilation feature. Except for very large packages of routines, this should be sufficient for most needs.

Also, there are no program-overlay or -segment facilities. It is possible to compile a program of type "CHN" that can be called from another Turbo Pascal program with the **Chain** standard procedure. A "CHN" file does not contain a copy of the Turbo Pascal runtime package and is, therefore, smaller than a "COM" file would be. In addition, it is possible to share variables between the calling program and the program chained to. You can use the **Execute** procedure to start execution of a "COM" file from a Turbo Pascal program. Both of these procedures place a value of FF (hex) in the command-line buffer as a means of notifying the called program that it has been invoked from another program. Because of this, calls should not be made to programs that expect to make use of information from the command line.

The use of external procedures is a little tricky in Turbo Pascal. It doesn't appear to be possible at all with programs compiled in memory. With a minor deviation, external procedures and functions are declared in a manner similar to the declaration of a **forward** subroutine; the procedure or function heading is specified, followed by the **external** reserved word. Following the **external** reserved word, there must be an absolute address for the routine.

Turbo Pascal assumes that all **external** subroutines will be written in assembly language. The reference manual clearly explains the parameter-passing and function-return protocols needed to make such assembly-language routines work with Turbo Pascal.

When the program is compiled, the starting address must be set explicitly from the Options menu to make room for the external subroutine between the runtime library and the rest of the program. I could find no way to get such a subroutine into memory at the address specified while the Turbo package was running. Apparently the only way to do so is to compile the program to a "COM" or "CHN" file, then – using the system debugger – overlay the external routine at the appropriate address in the code file and save the altered program. I attempted this with a simple external lowercase-to-uppercase conversion **function** and it worked. The process is probably a little complicated for a novice programmer.

Conclusions

The system is easy to install. This phase of getting a new program to work often frustrates users, but Turbo Pascal handles the process automatically if you have one of the approximately 30 terminals that it knows about. If you don't, a detailed installation procedure is described in an appendix of the users' manual, allowing installation on any modern CRT terminal. I tried both methods and had no problems.

As initially installed, the command set for the editor is very similar to a subset of WordStar. If you wish, you can change these commands.

Once installation is complete, using the system is a real pleasure. The integration of the compiler with the full-screen editor makes program development a snap. The compiler is incredibly quick, especially when translating a program for immediate execution in memory. The degree of interaction is very nearly that of working with a *BASIC interpreter*. If a compilation error occurs, the system will take you to the point in the program text where the error was detected. If an error occurs later, while running a program that was compiled in memory, the system will again take you to the source statement that caused the error. If an error is detected while running a "COM" or "CMD" file compiled by Turbo, it is also possible to find the source statement causing the problem.

Another important feature of Turbo Pascal is its size. It is small, both in terms of disk space and memory space. Although I did not try this, it appears that it would be possible to develop and run useful programs on computers with RAM memory as small as 32K.

The compiler accepts almost all standard Pascal statements. It is therefore possible to write highly portable programs. I had little trouble transferring programs between Turbo Pascal and UCSD Pascal. In many cases, no changes were required to compile and run the programs on either system. When changes were necessary, they were *usually simple* and straightforward. In most instances, the quality of the code produced was good, comparing favorably in size and speed with other Pascal and C compilers that I use on my system.

Turbo Pascal also has a large number of available extensions. Some allow easy access to low-level hardware and software functions of the host computer. Others include groups of built-in routines that are useful in many programs. Programmers often create libraries containing routines similar to these anyway, such as ones allowing control of the video terminal screen, but the Turbo system has already implemented many such facilities. Programs that use these extensions will probably not be directly portable to non-Turbo environments, but they are convenient.

The system is not without a few shortcomings, however. The absence of **Get, Put,** and **Dispose** may prove clumsy for some programs. Turbo Pascal is not acceptable for those applications that absolutely require separate compilation and overlay facilities. The alterations needed to include such capabilities in Turbo Pascal would probably negate many of its more desirable qualities. Also, the MicroCalc program, included as a programming example, requires a little polishing, mostly updating comments to take into account changes made in the program sections they refer to. Finally, the reference manual really

(Continued on page 109)

BOOK REVIEWS

High-Tech Consulting

By John Zarella
Published by Microcomputer Applications, November 1983
$18.95 paper, 167 pages
Reviewed by Robert Clark

Consulting is one of the most attractive and potentially lucrative ways for a hardware or software professional to establish an independent enterprise in the field of computer systems. A consulting business can be started on a shoestring, and can be initiated and run effectively by a single person. The business may begin returning profits almost at once, as high-tech companies will pay handsomely for the expertise needed to design new products using the latest technologies.

For those eager to capitalize on the opportunities presented by becoming a consultant, Zarella has some sound if sobering, advice on the business aspects of consulting. There are at least as many pitfalls as there are gold mines in this field, and anyone seriously considering starting out on his or her own will appreciate the pointers offered in *High-Tech Consulting*. Zarella spends little time recounting the attractive aspects of consulting; if you finish this book with as much enthusiasm for becoming a consultant as you had when you started it, your business will probably be extremely sucessful.

Part one of the book starts by assuming professional technical competence and describing other qualifications necessary for consulting. These include the abilities to listen to clients and analyze their needs, to estimate the time and materials necessary to complete a project, and to produce proposals and provide progress reports and other documentation on the work being done. Verbal communication skills are also needed to convince potential clients that you are the person for the job, and to ensure that they understand your approach to the problem. The latter includes conducting design reviews and explaining your work to technical and non-technical clients.

Zarella goes on to examine several notions about the consulting lifestyle. He admits that there are some advantages to being your own boss, selecting the jobs on which you will work, setting your rates, and managing your business and personal finances. He is quick to point out, however, that your source of income is always at risk, that your periods of free time will be unpredictable, and that your credit rating will probably disappear.

The second part of the book deals with setting up a consulting business. Such a business may be anything from an after-hours income supplement to a corporation with sizeable office space and many employees. The book is geared toward the one-person enterprise, and briefly covers such things as estimating startup costs, writing a business plan, obtaining a business license, paying taxes, recordkeeping, finding part-time or full-time employees, and dealing with Government red tape. (Think twice before taking a Government contract; filling out forms in triplicate can wipe out profit margins.)

In part three, the day-to-day aspects of running a consulting business are discussed. As a consultant, you will find that substantial energy will go into marketing your abilities, meeting with prospective clients, writing proposals, and other non-billable activities. Setting your rates and bidding on fixed-price jobs must be done with this in mind. These topics are covered along with suggestions on keeping technical records.

Zarella finishes up in part four with the topic of protecting yourself, your family, and your business with contracts, collection schedules, legal advice, patents and copyrights, and insurance, an Appendix of references and selected reading provides additional "how to" sources for the topics covered in this book.

If you follow all of the suggestions in *High-Tech Consulting* for establishing and maintaining your enterprise, you will have a model small business. You will also find yourself swamped in the details of running it. This book is valuable in that it offers advice on virtually every business aspect of consulting, but a one-person business can selectively disregard some of these. For instance, everyone should know what they are committing to when they sign a contract or non-disclosure agreement, but not everyone needs a business plan. Too much paperwork can make one lose sight of the fact that high-tech consulting is the business of hardware and software development, and a successful consultant should be able to enjoy doing just that if sound business practices are followed.

How to Create Your Own Computer Bulletin Board

By Lary L. Myers
Published by Tab Books Inc.
$12.50 (paper), 214 pages
Reviewed by James Moran*

Although no new ground is covered in *How to Create Your Own Computer Bulletin Board*, Lary Myers has done an admirable job of compiling program code for this book.

Designer of the FORUM80 and TABLOID BBS (Bulletin Board System) in Albany, New York, Myers has written a book that will be particularly useful to computer clubs that are interested in setting up their own boards. The first few chapters of the book serve a dual purpose in that they provide a technical introduction to the programs that follow as well as suggest some preferred BBS designs.

Although a little light on the narrative side, the book covers the basics of bulletin board design and does a good job of pointing out the danger areas in running these boards. In particular, serious BBS designers will appreciate the section on board security and integrity.

The majority of the book is taken up by BASIC and assembler programs that fully document a functional board for Apple and TRS-80 users. With the notable exception of the LTERM intelligent terminal program for TRS users, all of the assembler listings are nicely commented and should prove simple to modify.

The only serious problem with this book is the print quality of some of the listings. Whether this was due to the dot matrix printer that the author used to reproduce the listings or to a quality control problem at the printers is difficult to determine. In any case, for those who are interested, the publisher offers to supply the BBS programs on disk for $30. In view of the fact that the book is in large part a 160-page compendium of program listings, those BBS enthusiasts who are considering the implementation of their own board might want to bypass the book and just order the program diskette.

DDJ

*Copyright © 1984 CompuSyn.

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Professional Basic

Neil Bennett's outstanding BASIC implementation for the IBM PC, called "Professional Basic," is now available through Morgan Computing Company, Inc. (10400 North Central Expressway, Suite 210, Dallas, TX 75231). This version of the language sets new standards with its fantastic window-oriented debugging tools, dynamic syntax checking, use of the Intel 8087 numeric co-processor for fast floating-point operations, and support for the 8086/88 CPU's full 1-megabyte address space for both programs and data. The user interface has a distinctive, elegant style that is unbeatable for program development. For more details, read the review in the April 1984 issue of *Byte* magazine. If you own an IBM PC, and must program in BASIC for whatever reason, you owe it to yourself to pick up the phone and order a copy of Professional Basic. Users of other machines take heart; versions are being contemplated for most of the popular 16-bit microcomputers including 68000-based machines.

The Filepath Utility

"Filepath," a small but indispensable utility from SDA Associates (P.O. Box 36152, San Jose, CA 95158), has changed my life for the better. Written by Bernie Belew and selling for the miniscule fee of $12.95 plus tax, Filepath does for data files what the MS-DOS "Path" command does for executable program files. This is extremely helpful when using programs that refer to the disk for overlays, messages, and the like. For example, you can now put one copy of WordStar and its overlay files into a single subdirectory on your hard disk, point to that subdirectory with "Path=" and "Filepath=" specifications in your AUTOEXEC.BAT file, and use the word processor successfully from anywhere in the directory structure.

Beating the Laws of Thermodynamics

Those of you with a taste for computing esoterica might like to read the article "Computing without Dissipating Energy," in *Science* (March 16, 1984, page 1164). When I was a chemistry major, the Three Laws of Thermodynamics were taught to us as: (1) You can't get something for nothing, (2) You can't break even until you get to absolute zero, and (3) You can't get to absolute zero. There is now a heated (if you'll excuse the expression) debate going on between one group of physicists who believe computation inherently requires the expenditure of energy and some scientists who think the energy dissipation per logical operation could be brought arbitrarily close to zero with radically different technology. If you're interested, brush up on your Turing machine theory and lose yourself within the pages of *Science*.

Floating-Point Benchmarks

Results continue to flow in on the Bill Savage floating-point benchmark recently published in this column. In spite of its deficiencies, this benchmark seems to have sparked an incredible amount of reader interest. Versions for two of the "missing" languages, LISP and Logo, have been contributed and may be found in Listings One and Two (page 83). I plan to publish a corrected and expanded list of timings and errors within the next three months.

Mark Seage of Lawrence Livermore Labs came through with results for the Cray I and Cray X-MP processors and certainly made me eat my words about the performance of 8086/8087-based microcomputers compared to mainframes. The timing for the latter, incidentally, was an incredible 0.00012 seconds with an absolute error of 1.7E-10. He notes that this particular benchmark cannot be vectorized, but that in a benchmark that *was* vectorizable an additional speedup of 10 to 100 times could be expected. Now that I am confronted with an example of the number-crunching performance of a *real* mainframe that I can relate to, I must admit to being a bit awed.

At the other end of the computer spectrum, Jim Benenson wrote to defend the reputation of the Sinclair ZX-81. This machine's BASIC was previously reported to complete the Savage benchmark in about 1 day. Jim observed an execution time of 935 seconds in "fast mode" and an absolute error of 3.0E-1.

An error occurred in my transcription of the result for WS Systems Forth running on the 8086 with 8087 support. The actual error was 2E-13, not 1E-3 as printed in the column.

In the course of their comments on the original benchmark program, several readers have recommended *Software Manual for the Elementary Functions*, by William Cody Jr. and William Waite (Prentice Hall Inc., 1980). The book sells for about $20, and is "designed to help a programmer without a degree in numerical analysis implement the elementary mathematical functions," says Delbert Franz. It gives approximations for a variety of precisions and includes detailed flow charts and notes for a variety of floating-point bases ...as well as for fixed-point machines."

More on the Microsoft 8086 Assembler

My earlier column on some "anomalies" in the Microsoft Macro Assembler provoked a few indignant letters and some reports of further bugs. Several people came to the defense of the Assembler, saying that some of the problems that I considered bugs were simply artifacts of its internal storage format for variables. One person even stated that the Microsoft Assembler's display of word values in the opposite byte order from the actual physical storage in memory was "not a bug, but a feature!"

Sorry folks, I just can't agree. When you are using an assembler, you are working at the machine level and can expect to be staring at plenty of object code dumps while debugging. If the listing doesn't correspond *exactly* to what's in memory, it's only adding to your mental workload and creating unnecessary opportunities for additional errors. The same argument applies to the representation of the actual binary value of equates. If Microsoft wants to promulgate a philosophy of writing assemblers that is different from everyone else in the world, it could at least warn us in the manual.

Dan Rollins, author of an excellent assembly language column in *PC Age* magazine, writes, "There is another very nasty bug [in the Microsoft Assembler] that can cost hours of debugging. That is, when the symbol in

```
ADD  reg/mem16,offset symbol
```

has a segment offset of less than 128, the assembler generates the sign-extended version of the addressing mode byte. This is very considerate – it saves an entire byte in the executable module. However, the reference that's generated is not relocatable. Therefore if the position of the specified data changes during the link process, you end up with a logic error that's not your fault. The same problem exists for SUB, AND, CMP, OR, etc., when the address operand just happens to be early in the module.

"Here's a handy hint for your readers

Sometimes when you link a bunch of modules, you want to be sure that their segments will be loaded into memory in a specific order. For instance, you may want your DATA_SEG to come after your CODE_SEG. Alas, the physical placement of the segment in the source code has no effect on the placement in memory. The undocumented way to do this is to name your segments so that they are alphabetically in the order that you want them in memory. But, this only applies to segments in a single module.

"The Linker documentation clearly states that the overriding priority of segment placement is the order in which the object modules are named in the LINK command line. You can let that foul you up, or you can use it to your advantage.

"First, create several OBJ modules that each define an empty segment. For instance, one module would look like this:

```
; ** dummy module DATA_MOD.OBJ
data_seg segment public
data_seg ends
        end
```

"DATA_SEG would be the same name as a segment in the module you want to reorganize. Then have the linker load these 'dummy modules' before loading the real code:

```
LINK code_mod +stack_mod
    +data_mod +myprog;
```

"Rearranging the order of the dummy modules also rearranges the placement of the modules in memory. Verify it by reading the link map.

This technique is most useful when using the GROUP pseudo-op for creating COM format files. Often you'll want to define your data at the top of a program in order to avoid the phase errors and 'Operand must have size' errors that can happen when you have forward references in your code."

Page 5-40 of the Microsoft Assembler manual alludes to another way of controlling the placement of a segment. The "MEMORY" combine-type entry on a segment declaration is supposed to force that segment to be loaded at a higher address than all other segments being linked together. However, it doesn't work as advertised – just another "feature," I guess.

One particularly amusing report came from a user who found that mistyping "para" as "par" in a segment declaration can cause the Microsoft Assembler to crash without any warning or error message. The mechanism by which this could be occurring is hard to imagine. After all, the source file is, from the assembler's point of view, simply a data structure to be transformed, right? It's a good thing our word processors don't crash every time we misspell a word!

Toolbox Installment

Listing Three (page 84) contains a set of three interdependent subroutines to convert a word or byte into the equivalent hexadecimal ASCII representation. The routines are written in a form suitable for the Microsoft 8086 Assembler, though the code should also be acceptable with little or no change to Digital Research or Intel assemblers.

■■J

16-Bit

Listing One

Logo version of the Savage benchmark, adapted from program contributed by Tom Prince.

```
to savage ;Savage benchmark in LOGO
make "a 1
repeat 2500 [make "a (tan atan exp ln sqrt :a*:a 1) +1]
;note that LOGO atan is like FORTRAN atan2
(print [a = [ :a)
end

to tan :x
output (sin :x)/(cos :x)
end
```

End Listing One

Listing Two

LISP version of the Savage benchmark, contributed by Morton Goldberg.

```
~
~ An IQ LIST version of Savage's Benchmark Program
~
(DEF 'BMRK
  '[LAMBDA (N)
      (OR N (SETQ N 2500))
        [PROG (A)
           [SETQ A 1.0D)
```

(Continued on next page)

Listing Two

```
        LOOP
          (SETQ A (+ 1.ØD
            (TAN (ATAN (EXP (LN (SQRT (* A A ) T) T) T) T) T) T)))
          (SETQ N (SUB1 N))
          (COND [(LE N Ø) (RETURN (SUB1 A)])
          (GO LOOP)]])
```

End Listing Two

Listing Three

```
;
; Intel 8086 subroutines to convert a binary
; word or byte to hexadecimal ASCII format.
; Ray Duncan, March 1984.
;
word_to_hex proc near           ;convert 16-bit binary word
                                ; to hex ASCII
                                ;call with AX=binary value
                                ;          DI=addr to store string
                                ;             (4 characters)
                                ;returns AX, DI, CX destroyed
        push    ax
        mov     al,ah
        call    byte_to_hex     ;convert upper byte
        pop     ax
        call    byte_to_hex     ;convert lower byte
        ret
word_to_hex endp

byte_to_hex proc    near        ;convert binary byte to hex ASCII
                                ;call with AL=binary value
                                ;          DI=addr to store string
                                ;             (2 characters)
                                ;returns   AX, DI, CX destroyed

        sub     ah,ah           ;clear upper byte
        mov     cl,16
        div     cl              ;divide binary data by 16
        call    ascii           ;the quotient becomes the first
        stosb                   ;ASCII character, store it.
        mov     al,ah
        call    ascii           ;the remainder becomes the
        stosb                   ;second ASCII character, store it.
        ret
byte_to_hex endp

ascii   proc    near            ;convert bottom 4 bits in AL
        add     al,'Ø'          ;into the hex ASCII character
        cmp     al,'9'
        jle     ascii2          ;jump if in range Ø-9,
        add     al,'A'-'9'-1    ;offset it to range A-F,
ascii2: ret                     ;return ASCII char. in AL.
ascii   endp
```

End Listings

C/UNIX PROGRAMMER'S NOTEBOOK

by Anthony Skjellum

In previous columns, I have alluded to 8086-family C compilers that support large-memory models. Before discussing some of the compilers that provide such features, I thought it worthwhile to discuss the memory model concepts of the 8086 and how these concepts impact C compilers implemented for this microprocessor family. I will discuss the advantages and drawbacks of several memory models used by existing C compilers, and present code to help overcome some of the limitations of small-memory model compilers.

For those readers who are not interested in the details of 8086 C compilers, large-memory models, or long pointers, there is still some interesting material in this column. Specifically, several routines that are included here illustrate real-life code that will interface C and assembly language. Most compiler manuals are terse on this subject, so some actual code may help drive home the concepts involved.

Background

Before plunging into a discussion of memory models, a brief introduction to the 8086 architecture is necessary. This material will help to illustrate why different compilers use different addressing schemes.

The 8086/88 microprocessors support 20-bit addressing. This fact allows the microprocessor to address in excess of 1 million bytes. All the registers are 16 bits wide, however. This implies that some segmentation scheme must be used in order to address more than 64K of memory. The technique used involves four 16-bit segment registers: CS, DS, ES, and SS. These registers are the code-segment, data-segment, extra-segment, and stack-segment registers, respectively. Depending on the instruction used, different segment registers come into play in determining the complete 20-bit address. In forming a complete address, the segment address is always shifted left four bits. Note that a segment register by itself addresses memory on 16-byte boundaries. Sixteen-byte regions addressable by segment registers are known as paragraphs. Although paragraphs and paragraph alignment are not normally of interest to C programmers, they are sometimes important when developing assembly-language interface code for C.

When discussing long pointers, a special notation is used. Because the address is split, it is written in the form:

segment:byte_pointer

where *segment* is the segment, and *byte_pointer* is the 16-bit, low-order part of the address. A typical example of such an address would be "es:bx," which means "segment specified by es register and offset from this segment specified by the bx register." This notation is used throughout the listings included with this column.

Machine instructions often differentiate between inter- and intrasegment operations. For example, there are "near" and "far" CALL instructions.

I will now outline several memory models.

8080 Memory Model

The 8080 memory model is just what its name implies. All segment registers are set equal, so that only a total of 64K is available for a program. This model is seen mostly under CP/M-86 but is occasionally used by MS-DOS programs. None of the C compilers that I have seen restrict programs to this model.

Small-Memory Model

Many programs can work comfortably with only 64K of data space and 64K of program space. Such a model results when the CS and DS registers are set to different blocks of memory (up to 64K each). Normally, ES and SS are set equal to DS, so that all data and stack memory resides in the same block of memory. This model is fine as long as programs and data requirements are small enough to fit within the 64K limits. Most C compilers support only this model.

Large-Memory Model

In a large-memory model, all addresses refer to the full 20-bit range. All subroutine calls are "far" calls, and all data is referred to with long pointers. Long pointers include a segment- and byte-address pointer (thus occupying 32 bits). Only a few C compilers support this model. The reason that most compilers don't support this model is its complex code generation. I will mention more on this later.

Hybrid

A useful hybrid of the small- and the large-memory models is the one where only 64K of program space is provided but long pointers for data are used. This model offers speed advantages for programs that require more data storage but are moderately small.

One other possibility would be a large-code/small-data model, which would be used for programs with small data requirements but large code requirements.

One type of model that has not been considered is one that supports a large stack. A large stack would support more than 64K of items. Implementing this feature would slow program execution significantly, because stack references would be complicated.

Which Model Is Better?

As long as a C program can fit within the small-memory model, there is a distinct speed advantage in using this model. The large-memory model produces longer (and somewhat slower) programs because of the greater generality of each instruc-

```
typedef union   __lptr
{
        long    _llong;         /* long format */
        char    _lstr[4];       /* character format */
        LWORD   _lword;         /* long-word format */
} LPTR;
```

Figure 1.

```
typedef struct __lword
{
        unsigned _addr;         /* address */
        unsigned _segm;         /* segment */
} LWORD;
```

Figure 2.

tion produced (ability to refer to 1024K instead of 64K of memory requires longer pointers and more checks). Because the 8086 doesn't provide many instructions to manipulate the long pointers, many additional instructions must be generated for pointer-related operations (which also include all memory references).

Specific examples of the lack of 8086 instructions involve incrementing and decrementing long pointers. Note that a long pointer is not just a 32-bit word. The upper 16 bits is a segment address that must be treated accordingly when crossing 64K boundaries. Examples of implementing these features in software are included in Listing Five (11int. asm: examples: 1inc and 1dec functions, page 96).

Thus, both models have drawbacks. Speed is gained at the expense of (essentially) unlimited program and data space. Use the large-memory model for big programs that use big chunks of data. Otherwise, stick with the small model.

Drawbacks of the Small-Memory Model

Assuming that you use the small-memory model (by choice or because of your compiler), everything will run smoothly until it becomes necessary to deal with memory outside of the C data-address space. For example, it might be nice to use large buffers for copying files or for keeping help information. Another possibility would involve accessing special locations in the memory map.

The ability to use long pointers in a small-memory model can be implemented with relative ease. A set of such routines is presented in Listings One through Five (pages 88-104). A description of the Long Pointer Package and applications for the package form the remainder of this column.

The Long Pointer Package

The Long Pointer Package supplements a C environment by allowing references to memory locations to occur anywhere in the 20-bit address map. This is done by defining a new data type, LPTR (via a typedef), as shown in Figure 1 (page 86) where LWORD is defined as the structure shown in Figure 2 (page 86). This format for LPTR makes the addresses defined directly compatible with normal long pointers used at the assembly level. These long pointers are stored in the 8080 style: least-significant byte of address first, most-significant byte of segment last.

The lowest-level routines that support long memory references are, of necessity, coded in assembly language. The routines that implement many of the lowest-level functions in a noncompiler-specific way are included in Listing Four (11sup.asm). Routines that implement functions for Aztec C86 (a typical 8086 C compiler) Version 1.05i are included in Listing Five (11int.asm). The user may have to modify these routines to work with other C compilers if register usage or stack arrangements differ.

In order to actually use the routines with C programs, the header file "1sup.h" must be included at the beginning of modules that use or refer to LPTR data types. The "1sup.h" file refers to "_1sup.h" also. These two headers are presented in Listings Two and Three, respectively.

Supported Functions

The package supports a number of functions involving long pointers. There are routines to add offsets to long pointers and to copy memory between long pointers and routines to return data addressed by long pointers. A complete list of these functions is included in Table I (below). In this table, the file in which the function is located is mentioned.

An Example

One useful application of long pointers under MS-DOS 2.0 involves accessing a program's environment block. The environment block is a Unix-like set of environment variables and values. This is normally used to affect some particular aspects of program execution. Specifics about the environment address are included in the inset on page 88. Interested readers should also refer to the DOS 2.0 users' manual for more details.

The example program env.c reads the environment block and displays the contents of the whole block on the console. In effect, it provides the same listing feature as the MS-DOS SET command.

Conclusion

In this column I have discussed various aspects of memory models for 8086 C compilers. I have included a set

file: lsup.c (some C support routines)

lassign(dest,source)	assign long pointers
llstrcpy(dest,source)	long string copy
lprint(lptr)	debugging routine for printing LPTRs

file: llint.asm (Aztec C dependent support routines)

flptr(lptr,sptr)	form a long pointer from a normal short C (ds relative) pointer
lchr(lptr)	return character addressed by long pointer
lint(lptr)	returns int/unsigned addressed by long pointer
l_stchr(lptr,chr)	stores char at location lptr.
l_stint(lptr,intgr)	stores int at location lptr.
lload(dest,lptr,len)	general purpose copy to short pointer area (ds relative) from long pointer area
lstor(lptr,src,len)	reverse if lload( )
linc(lptr)	increment long pointer
ldec(lptr)	decrement long pointer
ladd(lptr,offset)	add unsigned offset to lptr
lsub(lptr,offset)	subtract unsigned offset from lptr
lsum(lptr,offset)	add signed offset to lptr
lcopy(dest,src,len)	general purpose long to long copy (can copy up to 1024K of memory)

file: llsup.asm (compiler independent functions)

linc	increment a long pointer
ldec	decrement a long pointer
ladd	add an unsigned offset to a long pointer
lsub	sub an unsigned offset from a long pointer
lsum	add a signed offset from a long pointer
lcopy	general copy routine

Table 1.

Environment Block Address

C compilers under MS-DOS normally produce .EXE files. For .EXE files, a program-segment prefix is created by DOS 2.0 and higher. The segment address of this prefix is es:0 when the user program begins. At offset 002CH from this address is stored the segment address of the environment table. Only a segment is stored: the offset from the segment is again zero. Thus, the contents of es:2CH is the address of the environment block.

Normally, C compilers have a maintenance routine that is given control at the start of program execution. In Aztec C86, this routine is called $begin and is located in the calldos.asm module included with the compiler. The user must define an external variable in calldos.asm for the benefit of env.c for the segment address to be accessible as a long pointer. The procedure for this operation is detailed in the comments included in Listing Six (env. c, page 104).

Allocation of Memory

If a C program intends to use DOS memory allocation in conjunction with the long pointers, it must also be sure to shrink its memory allocation, using the MS-DOS SETBLOCK function. This is normally done in the initial maintenance routine of the C runtime system. In Aztec C, this must be done in $begin.

of C and assembly-language functions that supports long pointers under a small-memory-model environment. With this package, users can enjoy the best of both worlds: access to arbitrary amounts and locations of memory, while retaining the efficiency of short pointers for regular code and pointer operations. For users of compilers that only support the small model, this package allows access to features that were previously off-limits to 8086 C programmers.

DDJ

C/UNIX (Text begins on page 86)

Listing One

```
/*************************************************************************
*                                                                        *
*       lsup.c                              created: 25-Mar-84           *
*                                                                        *
*       long pointer support for small memory model 8086 C compilers.    *
*                                                                        *
*       version 1.00 as of 25-Mar-84                                     *
*                                                                        *
*       Copyright 1984 (c) Anthony Skjellum.                             *
*       All rights reserved.                                             *
*                                                                        *
*       This program may be freely distributed for all non-commercial    *
*       purposes, but may not be sold.                                   *
*                                                                        *
*       The routines contained here are designed to be portable to       *
*       a large variety of compilers.  Currently they have been tested   *
*       with Aztec C86 v 1.05i only.                                     *
*                                                                        *
*       Modules comprising this package:                                 *
*                                                                        *
*               lsup.c          this file.                               *
*               lsup.h          header/definition file.                  *
*               _lsup.h         lower level header for this file         *
*               llsup.asm       assembly language support (compiler      *
*                               independent)                             *
*               llint.asm       compiler interface code   (compiler      *
*                               dependent)                               *
*                                                                        *
*       Subroutines included here:                                       *
*       (those marked with an asterisk are only included if compiler     *
*        used lacks some preprocessor support feature)                   *
*                                                                        *
*                                                                        *
*                                                                        *
*************************************************************************/
```

Listing One

```
#include "_lsup.h"                /* header with definitions */

/*
        Special routines: Included only if compiler lacks one of
        several features.
*/

/* lassign(dest,source): assignment of type LPTR to the left */

#ifndef MSUBST

lassign(dest,source)
LPTR dest;
LPTR source;
{
        dest._llong = source._llong;     /* assignment */
}

#endif

/*
        General purpose routines:
*/

/* llstrcpy(dest,src):  copy null terminated strings between long ptrs */

llstrcpy(dest,src)
LPTR *dest;
LPTR *src;
{
        char chr;                        /* temporary */

        while(1)                         /* loop */
        {
                chr = lchr(&src);        /* get   a character */
                l_stchr(&dest,chr);      /* store a character */

                linc(&dest);             /* increment destination ptr */
                linc(&src);              /* and source pointer */

                if(!chr)                 /* we are done at eos */
                        break;
        }

}

/* debugging routines: */

lprint(lptr)
LPTR *lptr;
{
        printf("%lx",lptr->_llong);
}
```

End Listing One

Listing Two

```
#include "_lsup.h"

/* place any special function specifications (defined in lsup.c) here: */
```

Listing Three

End Listing Two

```
/*****************************************************************************
*                                                                            *
*       _lsup.h                              created: 25-Mar-84              *
*                                                                            *
*       a component of lsup.c                                                *
*                                                                            *
*       version 1.00 as of 25-Mar-84                                         *
*                                                                            *
*       Copyright 1984 (c) Anthony Skjellum.                                 *
*       All rights reserved.                                                 *
*                                                                            *
*       This program may be freely distributed for all non-commercial        *
*       purposes, but may not be sold.                                       *
*                                                                            *
*       This is a header/definition file which must be included              *
*       in any module which utilizes long pointers.                          *
*                                                                            *
*****************************************************************************/

/*
        compiler feature toggles:
        comment out any which don't apply to the compiler in use.
*/

#define MSUBST                  /* macro substitution supported */

/* typedefs */

typedef struct __lword
{
        unsigned _addr;         /* address */
        unsigned _segm;         /* segment */
} LWORD;

typedef union __lptr
{
        long    _llong;         /* long format (for assignments) */
        char    _lstr[4];       /* character format */
        LWORD   _lword;         /* long-word format */
} LPTR;

/* constants */

/* macros */

/* lassign(destination,source): effect assignment of type LPTR */

#ifdef  MSUBST
#define lassign(d,s)    d._llong = s._llong;
#endif

/* function specifications: */

char lchr();
```

End Listing Three

(Listing Four begins on page 92)

Listing Four

```
;
; llsup.asm
;
; a component of lsup.c
;
; Copyright 1984 (c) A. Skjellum. All rights reserved.
;
; version of 25-Mar-84
;
; This routine makes no assumptions about the behavior of the
; C compiler in use.
;
; all procedures are "near"
;

dseg    segment para public 'data'
dseg    ends

cseg    segment para public 'code'
        assume  cs:cseg,ds:dseg

;
; linc:         increment a long pointer by 1 byte
;
; expects:      es:bx with long pointer to increment
; returns:      pointer incremented.
; consumes:     es, bx, f, ax
;
        public  linc
linc    proc    near
        inc     bx              ; increment low part of word
        or      bx,bx           ; is it zero now?
        jnz     linc_exit       ; no, we are done
        mov     ax,es
        add     ax,1000h        ; another 64k of paragraphs
        mov     es,ax           ; store back to es
linc_exit:
        ret                     ; return
linc    endp

;
; ldec:         decrement a long pointer by 1 byte
;
; expects:      es:bx with long pointer to decrement
; returns:      pointer decremented.
; consumes:     es, bx, f, ax
;
        public  ldec
ldec    proc    near
        or      bx,bx           ; zero currently ?
        dec     bx              ; decrement it
        jnz     ldec_exit       ; just decrement low end and exit...
        mov     ax,es           ; get segment register
        sub     ax,1000h        ; remove 64k of paragraphs
        mov     es,ax           ; store back to es
ldec_exit:
        ret                     ; return
ldec    endp
```

```
;
; ladd:         add a constant to a long pointer
;
; expects:      es:bx with long pointer's original value
;               ax with unsigned constant to be added
; returns:      pointer with constant added
; consumes:     es, bx, f, ax
;
        public  ladd
ladd    proc    near
        add     bx,ax           ; add in offset
        jnc     ladd_exit       ; no carry, so we are done.
        mov     ax,es
        add     ax,1000h        ; add 64k of paragraphs
        mov     es,ax           ; and store back to es
ladd_exit:
        ret
ladd    endp

;
; lsub:         subtract a constant from a long pointer
;
; expects:      es:bx with long pointer's original value
;               ax with unsigned constant to be subtracted
; returns:      pointer with constant subtracted
; consumes:     es, bx, f, ax
;
        public  lsub
lsub    proc    near
        sub     bx,ax           ; subtract offset
        jnb     lsub_exit       ; no borrow, so we are done.
        mov     ax,es
        sub     ax,1000h        ; subtract 64k of paragraphs
        mov     es,ax           ; and store back to es
lsub_exit:
        ret
lsub    endp

;
; lsum:         add a signed offset to a long pointer
;
; expects:      es:bx with long pointer
;               ax with signed offset
; returns:      pointer with constant added (signed)
; consumes:     es, bx, f, ax
;
        public  lsum
lsum    proc    near
        or      ax,ax           ; negative?
        jm      lsum_neg
        call    ladd            ; do addition
        ret                     ; and exit
lsum_neg:
        and     ax,07fffh       ; and out sign flag
        jnz     lsum_neg_ok
        mov     ax,8000h        ; -32768 value (don't treat as 0)
lsum_neg_ok:
        call    lsub
        ret
lsum    endp

;
; lcopy:        copy from one long pointer to another,
```

(Continued on next page)

Listing Four

```
;               up to 1024k bytes of data
;
; expects:      ds:si with src  address
;               es:di with dest address
;               ds!cx with length (dx is high order, cx is low order)
;
; returns:      block copied
;               ds, es intact
; consumes:     ax, cx, f
;
;
;       this routine uses a copy downward method, to produce
;       correct copying for overlapping regions
;
        public  lcopy
lcopy   proc    near
        ;
        ; convert dx into segment form:
        ;
        push    dx              ; save original form of dx
        push    cx              ; save low order of long count
        and     dx,15           ; smallest meaningful value
        xchg    dh,dl           ; switch upper and lower parts
        mov     cl,4
        shl     dh,cl           ; effect is shift left by 12 bits
        pop     cx              ; and recover low order of long count
        ;
        mov     ax,es
        add     ax,dx
        mov     es,ax
        mov     ax,ds
        add     ax,dx
        mov     ds,ax           ; gross adjustment of segments
        pop     dx              ; recover original form of dx
        ;
        add     di,cx           ; adjust dest. ptr to end of area
        jnc     no_dest_adj
        mov     ax,es
        add     ax,1000h        ; add offset
        mov     es,ax           ; and store back to segment register
no_dest_adj:
        ;
        ; do same work for source pointer:
        ;
        add     si,cx           ; do the addition
        jnc     no_mor_adj      ; no more adjustment needed if no carry
        mov     ax,ds
        add     ax,1000h        ; do the adjustment
        mov     ds,ax           ; and store back to ds
        ;
        ; at this stage:
        ;
        ;       es:di is at the last byte of the dest. area
        ;       ds:si is at the last byte of the src.  area
        ;
no_mor_adj:
        std                     ; set direction flag for moves
lc_loop:
        or      si,si           ; is si zero ?
```

```
        lodsb                   ; get byte ds:[si], decrement si
        jnz     no_ds_adj       ; no need to adjust if non-zero at start
        mov     ax,ds
        sub     ax,1000h
        mov     ds,ax           ; adjust pointer for next load
no_ds_adj:
        or      di,di           ; is di zero ?
        stosb                   ; set byte es:[di] = al, decrement di
        jnz     no_es_adj       ; no need to adjust if non-zero at start
        mov     ax,es
        sub     ax,1000h
        mov     es,ax           ; adjust pointer for next store
no_es_adj:
        loop    lc_loop         ; copy whole block (--cx, jnz lc_loop)
        dec     dx              ; work on outer loop
        or      dx,dx
        jnz     lc_loop         ; loop over dx counts too
        ;
        ; we are done
        ;
        inc     si
        inc     di              ; restore to original calling values
        ret                     ; exit
lcopy   endp

cseg    ends
        end
```

End Listing Four

Listing Five

```
;
; llint.asm
;
; version of 25-Mar-84
;
; a component of lsup.c
;
; Copyright 1984 (c) A. Skjellum. All rights reserved.
;
; these routines are setup for Aztec C86 v 1.05i
;
; all procedures are "near"
;

dseg    segment para public 'data'
dseg    ends

cseg    segment para public 'code'
        assume  cs:cseg,ds:dseg,es:dseg,ss:dseg

;
; Routines which do not merit calls to portable routines in llsup.asm
;
```

(Continued on next page)

Listing Five

```
;
; ds = flptr(lptr,sptr)
; LPTR *lptr;
; char *sptr;
;
; form a long pointer from a "normal" ds relative short pointer (sptr)
; and store at lptr
;
; note: no portable segment (flptr) in llsup.asm since this
; is such a trivial routine.
;
; return value is also ds, should this prove useful
;
        public  flptr_
flptr_  proc    near
        mov     bx,sp           ; prepare for argument load
        mov     ax,4[bx]        ; get short pointer ds:ax
        mov     bx,2[bx]        ; get address where to store long pointer
        mov     [bx],ax         ; store low order
        mov     ax,ds
        mov     2[bx],ax        ; store high order
        ret                     ; return value is also ds
flptr_  endp

;
; the following four routines are examples of what can
; be done to supplement general routines with specific
; (more efficient ones).  Many more variations are
; possible than the two presented here.  They follow
; directly from this basic idea:
;
;
; char lchr(lptr)
; LPTR *lptr;
;
; return character pointed to by lptr
;
        public  lchr_
lchr_   proc    near
        mov     bx,sp           ; prepare for argument load
        push    ds              ; save ds register
        mov     bx,2[bx]        ; get address of lptr
        mov     ax,[bx]         ;
        mov     ds,2[bx]        ; begin forming pointer
        mov     bx,ax           ; ds:bx now is valid pointer
        sub     ax,ax           ; zero whole acc.
        mov     al,[bx]         ; get the character
        pop     ds
        ret                     ; exit with char in ax
lchr_   endp

;
; int lint(lptr)
; LPTR *lptr;
;
; return integer or unsigned pointed to by lptr
;
        public  lint_
lint_   proc    near
```

```
        mov     bx,sp           ; prepare for argument load
        push    ds              ; save ds register
        mov     bx,2[bx]        ; get address of lpr
        mov     ax,[bx]         ;
        mov     ds,2[bx]        ; begin forming pointer
        mov     bx,ax           ; ds:bx now is valid pointer
        mov     ax,[bx]         ; get the integer or unsigned
        pop     ds              ; recover old ds value
        ret                     ; and exit with char in ax
lint_   endp

;
; l_stchr(lptr,chr)
; LPTR *lptr;
; char chr;
;
; store character chr at address lptr
;
        public  l_stchr_
l_stchr_ proc   near
        mov     bx,sp           ; prepare for argument load
        mov     cl,4[bx]        ; get character
        mov     bx,2[bx]        ; prepare for load of long pointer
        push    ds              ; save ds segment register
        mov     ax,[bx]         ;
        mov     ds,2[bx]        ; begin forming pointer
        mov     bx,ax           ; ds:bx now is valid pointer
        mov     [bx],cl         ; store byte
        pop     ds
        ret
l_stchr_ endp

;
; l_stint(lptr,val)
; LPTR *lptr;
; int val;
;
; store integer or unsigned at address lptr
;
        public  l_stint_
l_stint_ proc   near

        mov     bx,sp           ; prepare for argument load
        mov     cx,4[bx]        ; get integer to store
        mov     bx,2[bx]        ; prepare to form ds:bx with correct
                                ; storage address
        push    ds              ; save current ds
        mov     ax,[bx]         ;
        mov     ds,2[bx]        ; begin forming pointer
        mov     bx,ax           ; ds:bx now is valid pointer
        mov     [bx],cx         ; store the integer
        pop     ds
        ret                     ; exit
l_stint_ endp

;
; lload(dest,lptr,len)
; char *dest;
; LPTR *lptr;
; unsigned len;
;
; general purpose copy routine from long data storage to ds: relative
; storage
```

(Continued on page 100)

Listing Five

```
;
; we assume es = ds for Aztec C explicitly here
;
; due to convenient 8086 instructions, the portable function would
; consist merely of a "cld", "rep movsb" sequence followed by a return.
; Therefore, no portable lload is included in llsup.asm
;

        public  lload_
lload_  proc    near
        mov     bx,sp           ; prepare for argument load
        push    ds              ; save Aztec ds segment
        push    si
        push    di              ; save source and destination indices
        ;
        mov     cx,6[bx]        ; get length for move
        mov     di,2[bx]        ; es:di now has the destination address
        mov     bx,4[bx]        ; prepare to load long ptr
        mov     si,[bx]         ; get low order
        mov     ds,2[bx]        ; and then high order
        cld
        rep     movsb           ; do the move
        pop     di
        pop     si
        pop     ds
        ret
lload_  endp

;
; lstor(lptr,src,len)
; LPTR *lptr;
; char *src;
; unsigned len;
;
; Reverse of lload: this routine copies data from ds:src to lptr
; Once again, there is no llsup analog.
;
        public  lstor_
lstor_  proc    near
        mov     bx,sp           ; prepare for argument load
        push    es
        push    di
        push    si              ; save registers as required by Aztec C.
        mov     cx,6[bx]        ; get length of move
        mov     si,4[bx]        ; ds:si now contains source index
        mov     bx,2[bx]        ; prepare to form es:di
        mov     di,[bx]
        mov     es,2[bx]
        rep     movsb           ; move the data
        pop     si
        pop     di
        pop     es
        ret                     ; restore registers and exit
lstor_  endp

;
; ------------------------------------------------------
;
; routines which call portable subroutines in llsup.asm
```

```
;
; ------------------------------------------------
;
; linc(lptr)
; LPTR *lptr;
;
; increment a long pointer by 1
;
        public  linc_
linc_   proc    near
        extrn   linc:near
        mov     bx,sp           ; prepare for argument load
        push    es              ; save es value from caller
        mov     bx,2[bx]
        push    bx              ; address where answer will go
        mov     es,2[bx]        ; get segment
        mov     bx,[bx]         ; and address
        call    linc            ; do the work
        pop     ax
        xchg    ax,bx
        mov     [bx],ax
        mov     2[bx],es        ; store the value
        pop     es              ; recover the old value
        ret                     ; and exit
linc_   endp

;
; ldec(lptr)
; LPTR *lptr;
;
; decrement a long pointer by 1
;
        public  ldec_
ldec_   proc    near
        extrn   ldec:near
        mov     bx,sp           ; prepare for argument load
        push    es              ; preserve es
        mov     bx,2[bx]        ; get address ds:bx
        push    bx              ; address where answer will go
        mov     ax,[bx]         ;
        mov     es,2[bx]
        mov     bx,ax
        call    ldec            ; do the decrement
        pop     ax
        xchg    ax,bx
        mov     [bx],ax
        mov     2[bx],es        ; store the value
        pop     es              ; recover it for sake of caller
        ret                     ; and then exit
ldec_   endp

;
; ladd(lptr,offset)
; LPTR *lptr;
; unsigned offset;
;
; add unsigned offset to a long pointer
;
        public  ladd_
ladd_   proc    near
        extrn   ladd:near
        mov     bx,sp           ; prepare for argument load
        push    es              ; save es value of caller
        mov     ax,4[bx]        ; get the offset to ax
```

(Continued on next page)

Listing Five

```
        mov     bx,2[bx]
        push    bx                  ; address where answer will go too.
        mov     cx,[bx]
        mov     es,2[bx]
        mov     bx,cx
        call    ladd                ; do the addition
        pop     ax
        xchg    ax,bx
        mov     [bx],ax
        mov     2[bx],es            ; store the value
        pop     es                  ; recover old es value
        ret                         ; and then exit
ladd_   endp

;
; lsub(lptr,offset)
; LPTR *lptr;
; unsigned offset;
;
; subtract unsigned offset from a long pointer
;
        public  lsub_
lsub_   proc    near
        extrn   lsub:near
        mov     bx,sp               ; prepare for argument load
        push    es                  ; preserve es
        mov     ax,4[bx]            ; get the offset
        mov     bx,2[bx]
        push    bx                  ; store answer at this addr. too.
        mov     cx,[bx]
        mov     es,2[bx]
        mov     bx,cx
        call    lsub                ; do the subtraction
        pop     ax
        xchg    ax,bx
        mov     [bx],ax
        mov     2[bx],es            ; store the value
        pop     es                  ; restore es
        ret                         ; and then exit
lsub_   endp

;
; lsum(lptr,offset)
; LPTR *lptr;
; int offset;
;
; add signed offset to a long pointer
;

        public  lsum_
lsum_   proc    near
        extrn   lsum:near
        mov     bx,sp               ; prepare for argument load
        push    es                  ; preserve caller's es
        mov     ax,4[bx]            ; get the signed offset
        mov     bx,2[bx]
        push    bx
        mov     cx,[bx]
```

```
        mov     es,2[bx]
        mov     bx,cs
        call    lsum            ; do the signed addition
        pop     ax
        xchg    ax,bx
        mov     [bx],ax
        mov     2[bx],es        ; store the value
        pop     es              ;
        ret                     ; exit.
lsum_   endp
;
; lcopy(dest,src,len)
; LPTR *dest;
; LPTR *src;
; long len;     (treated as a long unsigned quantity)
;
;       copy from src to dest, len bytes.
;
; note this routine can be used to copy arbitrarily large chunks of memory
;
        public  lcopy_
lcopy_  proc    near
        extrn   lcopy:near
        mov     bx,sp           ; prepare for argument load
        push    ds
        push    es              ; save segment registers
        push    di
        push    si              ; save these registers for Aztec C
        ;
        mov     cx,6[bx]        ; get length (low order)
        mov     dx,8[bx]        ; high order of length
        mov     ax,2[bx]        ; get ds:ax as pointer to dest.
        xchg    ax,bx
        mov     di,[bx]
        mov     es,2[bx]
        mov     bx,ax
        mov     bx,4[bx]        ; get ds:bx as pointer for dest.
        mov     si,[bx]
        mov     ds,2[bx]        ; get long pointer
        ;
        call    lcopy
        ;
        pop     si
        pop     di
        pop     es
        pop     ds
        ret
lcopy_  endp

cseg    ends
        end
```

End Listing Five

Listing Six

```
/*
        env.c                                   created: 25-Mar-84
        This package echoes the environment to the standard output.
        example of using long pointers with lsup package.
        This is set-up to work with Aztec C.
        by Anthony Skjellum. (C) 1984. All rights reserved.
        Released for non-commercial purposes only.

        This program echoes the environment block to the console.
        In effect, this is the same as the DOS 2.0 SET command.
```

Listing Six

```
        Nevertheless, it illustrates the usefulness of long pointers.

        ------------------------------------------

        The following changes were made to the $begin
        routine of the Aztec C 1.05i module calldos.asm:

                i) a new global variable called envseg was created

                        envseg_ segment word common 'data'
                        $envdat dw 0
                        $envseg dw ?
                        envseg_ ends
               ii) On entry to $begin, when es contains the
                   program segment prefix (PSP), es:[2ch] contains
                   the segment address of the environment.  This
                   segment address is stored into the second word
                   of envseg_ (ie $envseg).

                   The environment may now be referred to through
                   the external LPTR envseg.
              iii) If DOS 2.0 allocation is to be used, be sure to
                   shrink the program size using the SETBLOCK function.
                   This must also be done in $begin where the psp,
                   ds, segments are both available.
*/

#include <stdio.h>
#include "lsup.h"       /* support for long pointers */

extern  LPTR envseg;    /* envseg is a structure of type LPTR */

main(argc,argv)
int argc;
char *argv[];
{
        char chr;
        int i;
        LPTR lptr;

        lassign(lptr,envseg); /* get long pointer to environment */

        while(1)        /* loop */
        {
                chr = lchr(&lptr);      /* get the next byte */

                if(!chr) /* we have hit the end of the environment */
                        break;
                while((chr = lchr(&lptr))) /* get characters of string */
                {
                        putchar(chr);     /* write them to console */
                        linc(&lptr);      /* increment pointer */
                }
                linc(&lptr);    /* pass the zero byte just encountered */

                putchar('\n');  /* add new line between entries */

        } /* end while(1) */

}
```

End Listings

OF INTEREST

by Michael Wiesenberg

Track Zero

I knew that this year's **West Coast Computer Faire** would be hoppin' from the moment I tried to get in on Saturday. "No problem," *DDJ* editor **Renny Wiggins** had said on Thursday. "Your pass is waiting for you. I saw it myself."

It seems that *DDJ* passes can be filed under four possible names: **Doctor Dobb's Journal, People's Computer Company, M & T Publishing, Inc.**, and **Business Software** (the new sister publication). I checked all with no success.

Officials kept shunting me from one line to another, from one person with the authority to grant tickets to another. They were less than impressed with my "business card" that Renny had assured me would be my entree if all else failed: the name of the magazine, the new address, but not my name.

In desperation I called the *DDJ* booth. New editor-in-chief **Mike Swaine** came out to assist. After 15 minutes of further fruitless official shuttling and line waiting, I asked Mike, "What would **Usasi** do?" "Sneak in the side door."

Finally **Renny** showed up. He found my ticket misplaced among the press passes, and I was in.

The show itself was a **bore**. There was nothing to match last year's **Perfect Software** laser and fog spaceship, nor the numerous **robots** walking among the throngs.

This year the most exciting offerings were several speech synthesizers that faithfully reproduced verbally anything passers-by typed on their keyboards. **Apple** had a 16-foot **Mac** mockup, but we've seen giant screens before, particularly those with **wavering, fuzzy images**.

Probably the **best show in town** was the **Friends of Dr. Dobb's Party** that night at the **Vorpal Gallery**.

The **Vorpal Gallery**, filled with world renowned **modern art**, including many **Escher** originals, was a classy place for **M & T** to announce its licensing agreement with **Dr. Dobb's Journal**.

The food was excellent – piping hot meatballs in sauce, vegetables with several dips, stuffed mushrooms, puff pastries, brie – and the many varieties of wine never stopped flowing.

I asked **Timothy Leary** if he was going to get into psychedelic software. He said that it was already here, that computers were becoming the **drugs** of the eighties.

I shook hands with **Robin Williams**, who was carrying a plastic shopping bag full of computer goodies and literature just like the rest of us.

Sat Tara Singh Khalsa, president of **Kriya Systems**, and devoted to the idea of **programmer as superstar**, held court in turban and white Sikh outfit.

Art Kleiner, editor of the **Whole Earth Software Catalog and Review**, was there, as were **Tony Bove** and **Cheryl Rhodes**, copublishers of **The Users' Guide**.

Dick Heiser, who started the first computer store ever, showed up, as did **Gordon Eubanks**, developer of **CBASIC**.

I spoke with **John Draper**, better known in his persona of the infamous **Cap'n Crunch**, phone phreak and publicizer of various rainbow-colored boxes. **Gerald Schmidt**, the undercover FBI informant who has been called " the most knowledgeable person on computer crime in the country," met Crunch face-to-face for the first time at the party.

Roger Gregory, systems anarchist, and **Ted Nelson**, both of **Project Xanadu**, floated about the canapes.

John Markoff, West Coast editor for **Byte**, and **Paul Freiberger**, who has the same position with **Popular Computing**, talked with their former associate, **David Needle**, news editor of **InfoWorld**.

Carl Helmers, the first editor of **Byte**, was seen, as were **David Ahl**, publisher of **Creative Computing** and **Sync**, and **Elizabeth Staples**, editor of **Creative Computing**.

Robert Harp, president of **Corona Systems**, wandered around.

Chris Terry, technical editor of **Microsystems**, was in evidence.

Midway through the party, **John Barry**, managing editor of **InfoWorld Books**, and **Jack Klyster**, mad inventor of the **klystron tube**, came in together.

While punching it down **Market Street** in my **Volvo station wagon**, listening to **Antonin Dvorak**'s *New World Symphony* on the stereo, I said to myself, "I hope they give me real business cards before next year's Faire."

Punching Diskettes

The **Disk Notcher** from Quorum makes another write enable notch so you can use the flip side of diskettes. For Apple II+ and IIe, **Certifix** formats the new side, examines the surface, locks out flaws so they can't be used, displays and saves status report, and initializes with DOS 3.3. Both products cost $29.95, with 64 write protect tabs and 32 diskette labels thrown in, or Certifix alone for $24.95, and the Disk Notcher tool alone for $15. Add $1.50 p. and h., and Californians add sales tax.

A Likely Case

Anvil Cases fit most microcomputers and are virtually indestructible. They are foam-lined with aluminum exterior. Some can be customized by users for unusual configurations. In addition to the portable cases, rackmount cases are available for heavier equipment. Anvil has been making these cases for 20 years for the music industry, to take sound systems on the road.

Dazzle Your Friends

One of the best graphics displays I saw at the Faire was produced by the **Graphics Dazzler** from Sigma Designs, a plug-in for the IBM PC, with 1024 x 1024 x 4 maximum display memory, 640 x 200 standard viewable display, 640 x 400 in "interlaced" mode. You get two display planes for four colors standard, or four display planes for 16 with the **Graphics Enhancer** stack. The board is compatible with all color and black-and-white monitors used with the PC. The display area can be panned over the 1024 x 1024 display memory, with a 1-to-16 zoom over any display area. All the work is performed by NEC 7220 display controller software, which also has hardware line drawing, area filling, and arc drawing. There is a light pen interface, and you get graphic display software and DOS

boot program. The Dazzler is $895, Enhancer $695. The **Maximizer**, at $395, has, in one card, 64K RAM, expandable to 384K, parallel interface for printer or bidirectional I/O, RS-232C serial interface, clock/calendar with battery backup, RAM disk, printer spooler, and, optionally, second RS-232C ($60) and a game control adapter that supports four paddles or two joysticks ($40). Sigma also offers various expansion cards, video cards, and hard disk subsystems that add from 10 to 33 Mb from $1695 to $4295.

Two More Processors for Heath/Zenith

The **H-1000**, from Technical Micro Systems, is a Z80/8086 plug-in replacement upgrade for H89/Z89 that needs no modifications. The Z80 runs at 2/4 MHz, and the 8086 at 8 MHz. The board comes with 256K RAM, expandable to 1 Mb. It has five I/O slots, and is fully compatible with all Heath/Zenith peripherals. It runs all Heath/Zenith software without modification, and is compatible with Zenith Z100 and IBM PC. You get your choice of MS-DOS or CP/M-86. TMSI seems a bit confused about its prices. The specifications brochure claims 256K RAM standard, while the price list cites the basic board *with 128K* for $995, and *upgrade* to 256K for another $250. Expansion to 512K is $1695, and to 1 Mb, $3495. They will also sell it to you as a complete system in a modified H89/Z89 case: 128K and one 100K disk drive, $2495; 256K and one 320K drive, $2995; or 256K and two 320K drives, $3295. TMSI also has a lot of software for the board and computer, including Concurrent CP/M-86, MP/M-86, RAM disk utilities, accounting and communications packages, data base, languages legal, planning, spreadsheet, word processing. You can also discuss their making a custom configuration for you.

Mass Marketing Software

Paul Terrell believes that software should be vended at K-Marts, Tower Recordses, Long'ses, and 7-Elevens on reprogrammable video game cartridges. The president of **Romox** was showing off his products at the Faire, including programs for as little as $7.95. When you get tired of one, just bring back the cartridge, pay the clerk from $7.95 to $15, and get a new one. Not just games, but education, business, etc., are available now for Atari computers and 2600, Commodore 64, Vic 20, TI 99/44A, and coming soon for IBM PC, PCjr, Coleco and MSX. I saw the ROM programmer at work. You can see a list of all games and get a short preview of any game on the list.

Get GUMMed

The **Gurus of Unix Meeting of Minds (GUMM)** takes place Wednesday, April 1, 2076 (check *that* in your perpetual calendar program), 14 feet above the ground directly in front of the Milpitas Gumps. Members will grep each other by the hand (after intro), yacc a lot, smoke filtered chroots in pipes, chown with forks, use the wc (unless uuclean), fseek nice zombie processes, strip, and sleep, but not, we hope, od. Three days will be devoted to discussion of the ramifications of whodo. Two seconds have been alloted for a complete rundown of all the user-friendly features of Unix. Seminars include "Everything You Know is Wrong," led by Tom Kempson, "Batman or Cat:MAN?" led by Richie Dennis, "cc C? Si! Si!" led by Kerwin Bernighan, and "Document Unix? Are You Kidding?" led by Jan Yeats. No **Reader Service No.** is necessary because all GUGUs (Gurus of Unix Group of Users) *already* know everything we could tell them.

Simply the Best

Computer Literacy Book Shop claims to be the best computer bookstore in the world, and the only one devoted exclusively to computers. They have over 4500 titles, and if you don't find what you want, they'll order it. Owners Dan Doernberg and Rachel Unkefer have acted as consultants to the **Whole Earth Software Review.**

Pay for It if You Like It

PC-Write, from Quicksoft, has the right marketing idea. This excellent word processor for IBM PC and PCjr (which some reviewers claim is better than the old standbys; it has all the features of the $400 word processors and a few more) costs $10 if you get it from the publisher, or it's free if you get it from a friend. Quicksoft encourages distribution of copies. They ask only that *if you like it* you send them $75. They'll send you a bound copy of their manual, give you telephone support, a free copy of the next revision, Pascal and assembly source, and a $25 commission when anyone else registers from a copy of your registered diskette.

Contact Points

Anvil Cases, Inc., 4128 Temple City Blvd., Rosemead, CA 91770; (213) 575-8614.

Computer Literacy Book Shop, 520 Lawrence Expressway, Suite 310, Sunnyvale, CA 94086; (408) 730-9955.

Quicksoft, 219 First N #224, Seattle, WA 98109; (206) 282-0452.

Quorum International, Unltd., Industrial Park Station, Box 2134, Oakland, CA 94614; (415) 552-8240.

Romox, Inc., 476 Vandell Way, Campbell, CA 95008; (408) 374-7200.

Sigma Designs, Inc., 2990 Scott Blvd., Santa Clara, CA 95050; (408) 496-0536.

Technical Micro Systems, Inc., Dept. H, 366 Cloverdale, Box 7227, Ann Arbor, MI 48107; (313) 994-0784.

DDJ

(Continued from page 77)

needs some more work to bring it up to the quality of the rest of the system.

In short, Turbo Pascal would be an excellent purchase for two groups of computer users: those who already know Pascal and those who don't. Programmers who already use Pascal will find it an exceptional program-development system for the vast majority of applications. Persons who want to learn Pascal should acquire a good textbook as well as the Turbo package. The highly interactive nature of Turbo Pascal will make it nearly painless to "learn by doing" by experimenting with examples in the textbook.

DDJ

p–A Small C Preprocessor

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

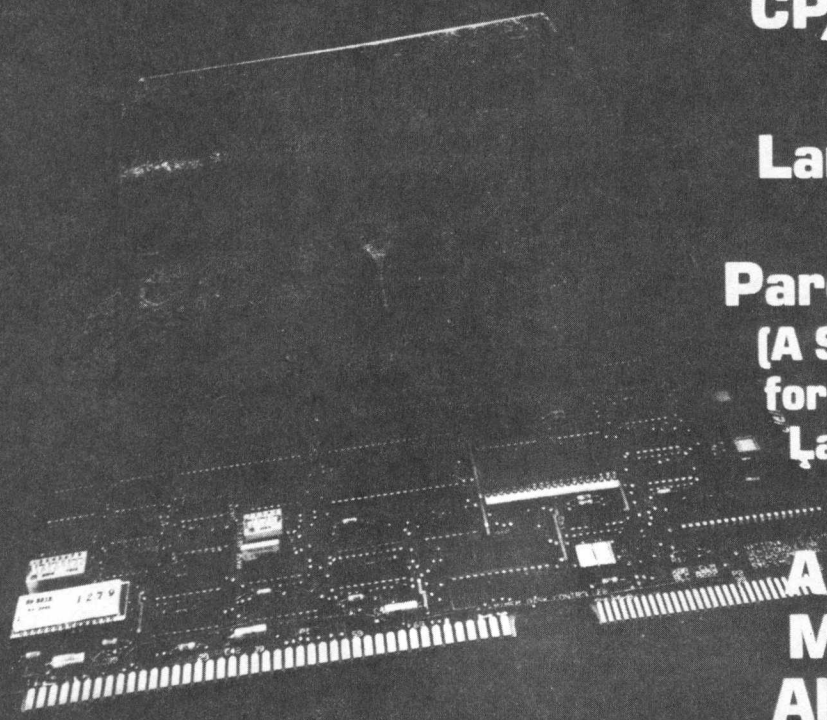
Dr. Dobb's Journal

#93 July 1984

$2.95 (3.50 Canada)

Resident System Extensions under CP/M Plus

Languages and Parentheses (A Suggestion for Forth-like Languages)



A Simple Minimax Algorithm

0 38351 16562 07

Dr. Dobb's Journal

Editorial

Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*

Production

Design Director *Fred Fehlau*
Art Director *Shelley Rae Doeden*
Production Manager *Detta Penna*
Production Assistant *Alida Hinton*
Cover Photo *Tom Upton*

Advertising

Advertising Director *Susan Cohen Strange*
Advertising Sales *Alice Hinton, Walter Andrzejewski*

Circulation

Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.

Publisher and Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, Box E, Menlo Park, CA 94025, a non-profit, educational corporation.

In This Issue

Our Resident Intern puzzled a few of months ago about the relative sparseness of CP/M Plus users. We're also surprised, in light of its greater flexibility and power than ordinary CP/M 2.2. Resident System Extensions are one example of this. This issue we have two pieces that deal with these handy routines. Garry Silvey of Digital Research has provided us with an article showing how they work and how to add them to your system. Bob Blum's CP/M Exchange column also received an RSX contribution from Mike Griswold that patches incompatibilities between CP/M Plus and 2.2 BIOS calls. Maybe this will give more folks incentive to explore what CP/M Plus can do for them. We plan to provide other material on CP/M Plus in the near future.

The Doctor's Bulletin Board

In the April issue we intimated that an electronic bulletin board could be in *DDJ*'s future. Well, what we have in mind is a multi-user system that will allow lots of features to be included. Because it is intended, among other things, to improve the reader interface, we would like to hear your suggestions of desirable capabilities. An excellent way to make your recommendations is the Editorial Response card (we even pay the postage). If you have something you would really like the board to do, jot it on the card with the any other comments you wish to make and drop it in the mail. We can't promise *everything* will be included, but we want it to serve as many needs a possible.

For Our Authors

Our latest version of the writer's guidelines is now available—expanded and accompanied by a current payment schedule. Just drop us a note with your name and address and we'll send them along to you. We are also adding ways to receive manuscripts. We can currently accept them via modem, MCI Mail, diskette in an increasing number of formats, and, of course, hard copy. You'll need to ask us about the particulars.

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to remarks to the editors concerning accuracy and relevance on manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness. Their remarks help prevent authors from exposing blindspots or misconceptions in print and help ensure that our readers receive clear and accurate information.

Numbering between forty and fifty referees, the referees include experts from diverse areas of the computer industry and the academic community. Because of space considerations, we can print a list of the entire board only a few times each year. Monthly, however, we do print the names of the referees who contributed their insights on material in that particular issue. Your humble editors must bear the burden of choosing how material ultimately appears, and we are grateful for the beneficial advice we receive.

The referees who contributed to this month's issue are:

Robert Blum, Contributing Editor, *DDJ*
David E. Cortesi, Contributing Editor, *DDJ*
James E. Hendrix, Office of Computing and Information Services, University of Mississippi
Richard G. Larson, Dept. of Mathematics, Statistics and Computer Science, University of Illinois at Chicago
William Ragsdale, President, Dorado Systems
Robert Smith

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

July 1984
Volume 9, Issue 7

CONTENTS

ARTICLES

36 Using Resident System Extensions Under CP/M Plus
by Garry M. Silvey
One feature of CP/M Plus is the ability to use Resident System Extensions—allowing a programmer to extend or modify any BDOS function, and to create new ones. This article discusses how they work and how to add them to your system.

46 *p* –A Small-C Preprocessor
by Axel T. Schreiner
Standalone preprocessors can be powerful tools in a variety of language settings. This independent preprocessor may be used with Small-C v2.0 and *cc*, the Small-C driver program presented last month.

84 A Simple Minimax Algorithm
by Steven A. Ruzinsky
While some of the classic minimax algorithms may be intuitively straightforward, they are often quite complicated to implement in a program. The author presents an automatic minimax algorithm which is relatively simple, easy to code, and yields a high degree of accuracy.

102 Languages and Parentheses (A Suggestion for Forth-like Languages
by Ernest E. Bergmann
Because many Forth-like languages do not need to use parentheses or similar characters as delimiters, this article proposes the optional use of such characters in these languages, discussing possible advantages and examining an actual implementation.

110 Balancing Act: The Ultimate Checkbook Balancing Program
by John E. Stith
The title says it all!

DEPARTMENTS

8 Editorial

10 Letters

17 Dr. Dobb's Clinic
The Missing RAM-drive; What Day Is This?; @In-, Out-, and Throughput; @Into the Lab

20 CP/M Exchange
Assembly language development packages and an RSX to patch CP/M 2.2's incompatibilities with CP/M Plus

112 Book Reviews
C Programmer's Library; Building Controls Into Structured Systems; MENTOR -- The Magazine on Disk

116 16-Bit Software Toolbox
iRMS-86 for the IBM PC; Concerning Redirection; C Programming Tools; Savage's Benchmark Again

122 Of Interest

EDITORIAL

There were programmers long before the invention of the digital computer in the 1930s and 40s. A century earlier, Jacquard's loom took its weaving instructions on punched cards: the person who selected the cards to produce the desired pattern was an early programmer.

Software designers came later. What Mauchly and Eckert and Atanasoff and Berry and Von Neumann and Turing and Zuse invented under the shadow of World War II was not just a programmable device, but a general-purpose programmable device, and that distinction made possible (and necessary) software designers.

A software designer makes the computer do something qualitatively different from anything it has done before, supplies the general-purpose device with a new specific purpose.

Without that definition of purpose, documentation is moot. Documentation for specific applications such as word processors and spreadsheet programs is getting better, but there has never been anything that can fairly be called computer documentation, no manual that explains, to someone who doesn't already know, what a computer does. One can't document the inherently undefined.

Without that definition of purpose, sales are problematic. Apple sales skyrocketed when Personal Software released VisiCalc for the machine.

So one adopts a simplifying metaphor. The computer is a spreadsheet, the computer is a desktop, the computer is a vending machine: drop in a problem and out falls a solution. That's how technical writers document, that's how dealers sell, that's how users understand computers. One needs that grounding metaphor, and it's the software designer who provides it.

The Xerox PARC designers wove their homely office-furnishings metaphors and produced the Alto and Star, whence Apple's hopes for its corporate future in starspawned Macintosh and Lisa. Dan Bricklin and Bob Frankston spun the metaphor of the computer as spreadsheet and came up with VisiCalc. A computer plus VisiCalc is an electronic spreadsheet, something whose purpose can be understood and documented.

But every metaphor is a simplification, a fiction which, believed, closes doors. Metaphors can be hidden inside the words we use, and can subtly influence our thinking. Is COPY really a better operating system command name than PIP? While COPY seems to say clearly what it does, PIP is an acronym whose full name, Peripheral Interchange Program, reminds us that it is a program, written by an individual for a purpose not necessarily coextensive with our intuition about what copying is. Which metaphor serves us better?

The ideal is to create good metaphors and not to be controlled by unexamined metaphors. That's what distinguishes a software designer from a programmer. The software designer is one who can glimpse the computer, however briefly, as a computer, as a truly general-purpose device, and then create a new purpose, a new metaphor. The good programmer uses an appropriate metaphor effectively; the software designer steps out in the undefined air and weaves a metaphor to stand on.

We think that good, creative software design is important, and we want to promote it. Within the next two months we will be introducing a new department of the magazine, The Software Designer, in which working software designers will discuss basic issues in software design.

Michael Swaine

A Small-C Redundancy

Dear Dr. Dobb,

I'm a new subscriber to *DDJ* who has recently managed to obtain a copy of Jim Hendrix's Small-C compiler. When I got it up and running, I discovered that some functions produced extraneous, unreachable code—namely, functions that explicitly returned a value as the last statement in the function and that also declared local variables. For example,

```
function() {
   char c;
   return(c);
   }
```

The problem was that the explicit return caused stack-adjusting instructions to be generated along with the RET, to delete the local variable space. These same instructions were generated again at the end of the function (which in this case was a couple of characters later, immediately after the return).

I traced the source of the problem to the compound() function and designed a patch, which I sent to Jim Hendrix. During the course of our correspondence on this matter, he came up with a better patch, which is contained in the material following this letter. I've applied the patch, and it seems to work as indicated.

I'm grateful to Ron Cain, J.E. Hendrix, and *Dr. Dobb's Journal* for providing this compiler. I learned a lot from it, and, now that I have it working, intend to make good use of it.

Thanks,
David Bookbinder
888 Mass Ave. #511
Cambridge, MA 02139

Mr. Hendrix writes:

David Bookbinder has observed that Small C generates redundant code for deallocating local variables at the end of a function when the last statement is a return. The same is true if a return is the last statement in any compound statement which declares local variables. This is harmless, but it unnecessarily adds to the size of programs and it really is untidy.

Likewise, if the last statement in a function is a goto, unnecessary code for deallocating local variables and a RET is generated. Since a goto would block the exit path from a function, there is no need for such code. (The compiler already eliminates the RET if the last statement is a return.) There is no problem with goto's terminating nested compound statements since goto's are not allowed if variables are declared anywhere after the opening brace of the function body. So there are no variables to deallocate.

The following patch will eliminate these instances of redundant and unnecessary code.

(1) Add the following line to the end of the file CC.DEF:

```
#define STLABEL 14
```

(2) Change the line in newfunc() (file CC12.C) which contains a call to statement() to read:

```
statement():
#ifdef STGOTO
if(lastst != STRETURN &&
lastst != STGOTO) ffret();
#else
if(lastst != STRETURN) ffret();
#endif
```

(3) Change the line in statement() (file CC13.C) which contains a call to dolabel() to read:

```
else if(dolabel())
lastst=STLABEL;
```

(4) Change the line in compound() (file CC13.C) which contains a call to modstk() to read:

```
#ifdef STGOTO
   if(lastst != STRETURN &&
   lastst != STGOTO)
#else
   if(lastst != STRETURN)
#endif
   modstk(savcsp, NO);
   /* delete local variable space */
csp=savcsp;
```

(5) Recompile, assemble, and link the compiler.

He Found Problems, Too

Dear Editor:

I read with considerable interest Ray Duncan's comments regarding the Microsoft Macro Assembler for the IBM PC in your February 1984 issue. Since the State University of New York at Buffalo is acquiring IBM PC's at rather a good clip, I decided to experiment a bit with the assembler under the theory that if there are that many bugs, there may well be many more. As it turned out, the first thing I tried failed. Basically, as the code in Listing One (page 12) demonstrates, the assembler seems to ignore the fact that IF-nesting is local to MACRO's (including IRP's, etc.) and *all* nested unsatisfied IFs must be cleared if the MACRO is exited via EXITM. If the seemingly "extra" ENDIF is removed (the line with the !'s), the assembler reports unclosed IF's on both passes. With the extra ENDIF, it assembles correctly, but this pattern of performance makes EXITM almost useless.

Further experiments with IRP's and EXITM's enclosed within MACRO's yielded results too bizarre to interpret simply or consistently. This situation, coupled with what Mr. Duncan observed, puts this piece of software firmly in the "Not Recommended" category insofar as SUNY at Buffalo is concerned!

I do disagree somewhat with Mr. Duncan's observation that a "more unlikely high level language" than Pascal could not have been chosen to write an

Listing One

Microsoft Assembler Illustration

```
The IBM Personal Computer MACRO Assembler 02-8-84  PAGE  1-1

                                ASSUME CS: CSEG
0000                    CSEG    SEGMENT
                                IRP  R,<B,C,D>
                                IFIDN <R>,<C>
                                DB    1
                                EXITM
                                ELSE
                                DB    0
                                ENDIF
                                ENDM
0000  00                +       DB    0
0001  01                +       DB    1
                                ENDIF  ; !!!!
0002                    CSEG    ENDS
                                END

Segments and groups:

                Name            Size    align   combine class
CSEG . . . . . . . . . . . . .  0002    PARA    NONE

Warning Severe
Errors  Errors
0       0
```

Listing Two

Accent Finder Fix

```
procedure prompt(var word:WordType; var Error:boolean);
  begin
    write('Enter the word > ');
    readln(word);
    word:=UpperCase(Condense(word));
    CheckChars(Error,word)
  end;
```

assembler in. After all, Microsoft could have chosen COBOL! (Such a choice is not quite as impossible as it might seem; back in the mid-1960's the French wrote part of an Algol compiler for the IBM 7040/44 series in COBOL.)

Seriously, Pascal is not a *bad* choice as language structures go; after all, both Modula and Ada descended from it. The problem is that the IBM PC Pascal compiler does not generate very good object code. This is a global microcomputer difficulty, as your Resident Intern, Mr. Cortesi, has been stating so eloquently in the last several issues.

Keep up the good work!

Sincerely,
Gary M. Gibson, Assoc. Dir.
University Computing Services
SUNY at Buffalo
4250 Ridge Lea Road
Buffalo, NY 14226

Accent Finder Fix

Dear Sirs,

It goes without saying that I was thrilled to see the "Accent Finder" in the May issue. In order to accommodate the changes you suggested, I had sent you three versions. The version published seems to be a combination of versions two and three. In its present shape it will work. However, it will fail to check for invalid characters within the word that's being analyzed. In order to remedy this, one need add only one line to the PROMPT procedure which should appear as shown in Listing Two (page 14).

For anyone interested in contacting our club, the UCLA IBM4341 Users' Group, we are on the BitNet. Our node is UCLAVM and my handle is VSS2364 (VSS2364 @ UCLAVM).

Long live Dr. Dobbs!

Yours truly,
Eddy C. Vasile
3314 Sawtelle 28
Los Angeles, CA 90066

More RSA Remarks

Dear Editor:

I am responding to the letters that you published in the June 1984 issue regarding my recent article on the RSA Public Key System.

First let me comment on remarks by Dr. Criscione. He points out that factoring of large prime numbers, in this case Mersenne primes, is being done. I pointed out in my article that a means of factoring large prime numbers would render the RSA system less useful. However, I believe that these primes are special cases and that the methods being used are specific to this set of primes. I should also point out that Knuth discusses methods of factoring primes in the book that I referenced in my article.

Next let me address remarks by Mr. Evenden. He states that RATFOR and FORTRAN are dinosaurs. Let me paraphrase Sam Clemens: The announcement of FORTRAN's death is premature. All languages have their pros and cons; FORTRAN, Pascal, C, etc., are no exceptions. I do not want to get into a debate over what language is best, but just want to say that languages are tools and any tool can be misused. I can just state that RATFOR/FORTRAN was what I had available at the time. Besides, FORTRAN is still the mainstay of the scientific and engineering community, good or bad.

He has submitted some C source that shows division/mod-less operations for the Multiple Precision Add/Subtract Algorithm implementation. I

(Continued on page19)

DR. DOBB'S CLINIC

by D.E. Cortesi

The Missing RAM-drive

In April we passed along Malcom Fordham's request for help. He uses CP/M-86 on an IBM PC and wanted to set aside some of the PC's storage for a RAM drive, but he couldn't find the software to do it.

Well, it turns out that the answer was right there in his own system. U. Thier of Bloomfield, CT, wrote to point out that CP/M-86 for the PC, as shipped by Digital Research, contains support for a RAM drive. We and Fordham had overlooked it because setting up the RAM drive is a function of the SETUP command. That command is documented only in the "Release Note," an unimposing little pamphlet at the back of the manual binder.

The Release Note looks like some kind of techie's afterthought, and quite a few users of CP/M-86 probably have never looked into it. That's unfortunate, because only there does one learn how to do such useful things as:

- Setting a command line that will be executed at power-up (you could set the line "submit profile," for example)
- Saving the configurations of the serial ports and function keys so you don't have to reset them after a cold boot
- Changing the disk step-rate, thus speeding up the system quite a bit
- Installing a special device driver

And, of course, setting the base address and size of a segment of storage to be used as drive M:. If you use CP/M-86 on the PC, take a close look at that Release Note and the SETUP command it describes.

What Day Is This?

This column started out three years ago (good grief, is it that long?) with a discussion of calendar algorithms. Apparently the message didn't get through to somebody (sigh, nobody listens to us), because Glenn Roberts of Knoxville, TN, has written to tell us of a calendar error in Lotus 1-2-3.

"I am always surprised how many sophisticated programmers do not really know the rules of the calendar," Roberts says. "Webster's New World Dictionary defines the Gregorian Calendar as follows:

a corrected form of the Julian calendar, introduced by Pope Gregory XIII in 1582 and now used in most countries of the world: it provides for an ordinary year of 365 days and a leap year of 366 days every fourth even year, exclusive of century years, which are leap years *only if exactly divisible by 400.* [emphasis added]

Thus the year 1900 was not a leap year (even though it is divisible by 4), but the year 2000 will be.

"The lack of understanding of this simple set of rules has shown up as bugs in some programs. Lotus 1-2-3, for example, contains a built-in function @DATE, which handles the years 1900 and 2000 incorrectly. For 1900 it thinks there *is* a February 29th, while for 2000 it thinks there *isn't* one—just the opposite of the truth. The internal representation of the date maintained by Lotus is supposed to be the number of days since December 31, 1899; however, it is actually one more than this for dates between 3/1/1900 and 2/29/2000."

Roberts goes on to say that it would be interesting to hear of other popular programs with calendar bugs. We think so, too.

@In-, Out-, and Throughput

R. C. Wagner, of Indecipherable, CA, wants to know what exactly we mean by "a throughput of x bytes per second," a phrase we used last November. It's a good question. We've never seen a formal definition but rather intuited the meaning of the word "throughput" from the contexts it appears in. Now that Wagner made us think about it, the idea has some holes.

Given a program whose input is from, and output is to, secondary storage (tape or disk) and given the need to say something meaningful about its total performance, you have two choices. You can talk about its elapsed runtime, but you always have to qualify that with the quantity of input (or output) data it processed in that run. You have to say something like: "It took 34 seconds to process a 73-kilobyte input file."

It's clearer to talk instead about the program's "throughput," meaning nothing more than the quantity of data it processed divided by the time it took to process it. That yields a figure of kilobytes per second. It's a simple measure and one that allows another person to project how long the program would take to process a file of some other size.

Well, maybe. When you really think about the idea, you realize that it rests on a lot of unstated assumptions that might not be true. Let's examine some of them.

First, throughput includes input time, output time, and all the processing in between. Since input, output, and processing all contribute to the total runtime, throughput measures the program's gross behavior. But it doesn't say anything about which of the three parts dominates. If the program is too slow (do we ever worry about a program because it's too fast?), knowing its throughput gives us no clue as to where to start looking for a bottleneck.

Second, if the program does little processing, its input and output speed will dominate its throughput rate. I/O speed is often determined by factors external to the program—the operating system, the device drivers, and the speed of the devices themselves. When

a throughput rate is quoted, the unstated assumption is that these external factors are constant. Of course, they are only constant within a single system not between systems (as we will see very shortly).

Third, there is an unstated assumption that the program runs in time that is a linear function of the quantity of data it processes; that is, if the amount of data is exactly doubled, the runtime will also exactly double. That's probably true of the I/O portions of the runtime but might be completely false as a description of the processing time. Programs exist whose processing time is *exponential* in the quantity of data they process. It would be quite deceptive to quote a throughput figure for such a program, since the "throughput rate" would change drastically with the amount of data—just the opposite of the accepted meaning.

Sort routines never have a linear processing time function, but that may not matter if sorting is only a small part of what a program does or if the sort time is very small relative to the I/O time. But many programs display nonlinear profiles when runtime is graphed against data quantity. Compilers, for example, may begin to spill their internal tables to disk at a certain size of source program, with the result that their runtime graph has a distinct "knee" at that size of input.

In short, "throughput" is a valid, if rough, measure of performance for programs that produce output that is proportional in quantity to their input and whose processing time is either insignificant or known to be a linear function of the amount of data processed. (That pretty well describes a typical utility program.) Even when these conditions are met, the problem of comparing the I/O environment remains.

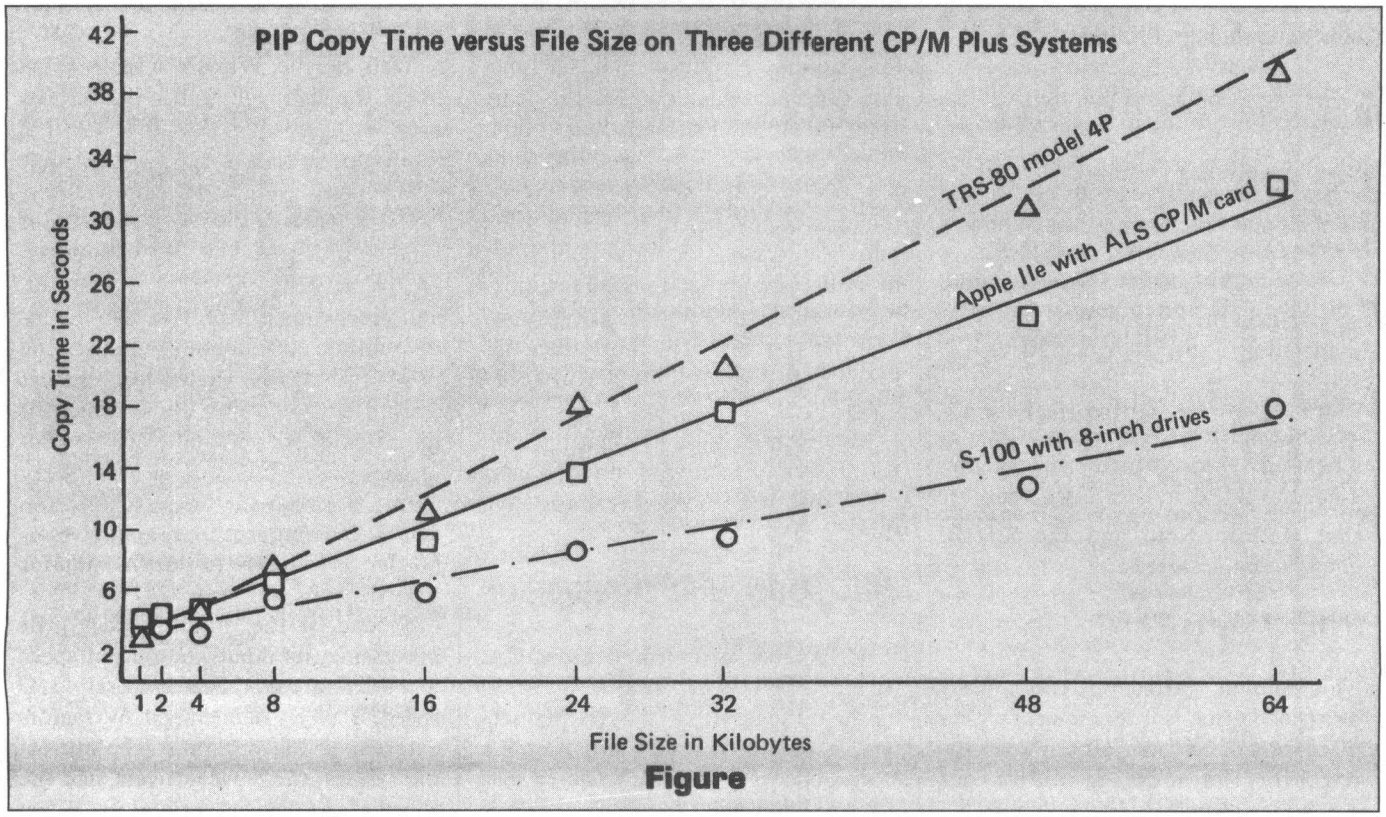
@Into the Lab

After writing that insightful commentary, we decided to see if we really knew what we were talking about. By good fortune, we have access to three systems that run identical software but have very different hardware. One is an S-100 system with 8-inch disk drives; one is a TRS-80 model 4P; and the third is an Apple IIe with Advanced Logic Systems' CP/M card. All three run CP/M Plus and have Z80 CPUs, but the Apple's CPU runs at 6 MHz while the others run at 4 MHz.

On each machine we set up a series of ASCII files of different sizes. Then we copied the files with PIP and measured the time it took to copy each file on each machine. The copies were from one drive to the other. Program load time was not measured; in each case PIP was called with no arguments, and only the copy time was measured. After each measurement the output file was erased, so every output file started at the same place on the output disk. However, the input files were built from smallest to largest, so they started at different tracks (the largest starting farthest from the directory).

The figure (below) summarizes the results. As you can see, each machine's times plot out as a good approximation to a straight line. (That verifies that hypothesis that PIP's processing time is a linear function of the amount of data it handles — no great surprise.) The slope of a line is the throughput rate for a PIP copy in that hardware environment.

The table (page 19) lists the measured times, along with the throughput rate that would be calculated from each measurement in isolation. Examine these numbers and you'll discover some pitfalls to measuring throughput. The calculated throughput for small files bears little resemblance to that for larger files. In fact, the linear functions



Figure

that show up so clearly in the graph aren't obvious in the numbers of the table at all. The best-fit straight line for each machine seems to approximate most closely to the calculated throughput for a 32-kilobyte file.

There's a lesson here. We've said before that computer science is one of the few that allows precise, repeatable measurements, and there really is no excuse for making relative judgments without having the measurements to back them up. We were guilty of exactly that last November, when we cited a "throughput" value based on exactly one measurement—tsk, tsk.

There's also an idea lurking in the figure. Disk I/O time dominates so many things we do with personal computers, it would be nice to have some kind of disk I/O figure of merit with which we could compare one system to another. The throughput rate of a simple file copy looks as if it might work for that purpose.

Where would your machine's line fall in the graph of the figure? Well, why don't you take the measurements and find out? Remember, program load time isn't included, just the time from entering the file transfer instruction until the next prompt. On a CP/M system, that's from pressing enter on "*b:=a:filename" through the next PIP asterisk prompt. On an MS-DOS system, it would be from pressing enter on "copy filename b:" until the next system prompt (since the MS-DOS copy command is built in).

If enough people send in measurements like those in the table, we'll publish an amalgamated graph. As a way of comparing day-to-day useful performance, it would be a bit more meaningful than the Sieve of Eratosthenes.

By the way, R. C. Wagner's question, which started all this, was written in the comment space on a Reader Service card. That's great! Any communications medium that gets more input for this column is okay by the Intern—although if you enter the First & Last Annual ZOSO Sound-Alike contest that way, you're going to have to write awfully small. DDJ

File size	S-100 system		TRS-80 Model 4P		Apple IIe/ALS	
(kilobytes)	time	thruput	time	thruput	time	thruput
1	3.4	0.29	3.2	0.31	4.0	0.25
2	3.4	0.59	4.0	0.50	4.2	0.48
4	3.4	1.18	5.0	0.80	5.0	0.80
8	4.8	1.67	7.2	1.11	6.4	1.25
16	5.8	2.76	11.0	1.45	9.2	1.74
24	8.2	2.93	17.4	1.38	13.8	1.74
32	9.6	3.33	20.6	1.55	17.4	1.83
48	13.2	3.63	30.0	1.60	23.6	2.03
64	18.0	3.56	39.5	1.62	32.5	1.97

Table.
The measured PIP copy times, from which the figure was drawn, and the calculated throughput rates that result

(Continued from page 12)

stated that all of the routines could certainly be improved. I thank him for his improvement and I welcome other suggestions.

He is correct that a typographic error exists in the Multiply Algorithm at step M5 (March 1984 *DDJ*, page 19). Instead of i=i–i it should read i=i–1. In addition, he is absolutely correct in a strict mathematical sense that INT(x) is not equivalent to FLOOR(x). However, as he states, it is true for non-negative integers (natural numbers). I pointed out that all of the mathematics of the RSA system would be limited to natural numbers, thus its equivalence in this case is valid.

Since I'm writing, I'll mention the few additional errors I have found. In Part I (March 1984), there is an underscore missing at the end of a line in Figure 1, page 17. The line (somewhat abbreviated) should read: "longline=variable1*(cos(. . .)+variable4* __". Also, in Figure 2b the "— or —" should have come *after* the line "= 16 mod 7 = 2", not before.

There are also three code alignment problems. In Part I, the nine lines of code at the top of page 36 (Listing Four) should be moved right one tab position so that the line "carry=0 . . ." aligns with the line "idxn2=len2 . . . ." Additionally, the top four lines on page 38 (Listing Four) should be moved right two tab positions so that the brace in line four of that page is indented from the "else" that follows it. In Part II (April 1984), the code block at the bottom third of page 57 (Listing Eight) which reads "do idx=1,8 # convert lower case to upper case" should be moved right one tab position to align with the line "read(CONSOLE,300) . . . ."

I hope this information helps.

Yours truly,
Charles E. Burton
1720 S. Deframe Ct.
Denver, CO 80228

DDJ

CP/M EXCHANGE

by Robert Blum

Searching for the right tools to fill your program development tool chest can be an exhausting experience; not to mention the damage to your wallet if you make any wrong decisions along the way. Fortunately, when looking for the right assembly language development package you will not have to contend with nearly as many competing packages as in other application areas. Nonetheless, deciding which package best suits your needs may be more difficult than necessary unless you first spend some time defining what your needs are and what those needs equate to in terms of dollars.

Before you go to the local software store to get a demonstration of the packages that they carry I suggest you first contact several manufacturers of packages that look interesting to better understand what you should expect to get for your money. I make this suggestion because I think you will find that very few software dealers know anything about the type of product you are looking for. And in most cases, even if you are fortunate enough to find a dealer that actually has this type of software available for purchase, you will probably have to read the manual to figure out how to run your own demonstration.

Getting Started

Most assembly language development packages fall into one of two categories. The under $50.00 package is aimed at the hobbyist or the infrequent assembly programmer. These packages are usually limited to an assembler that will accept Intel 8080 mnemonics and possibly extensions for the Z80. Occasionally one can be found that also offers the Zilog set of mnemonics for the Z80. On the opposite end of the spectrum is the commercial package priced at upwards of $150.00 or more. Packages of this type usually offer a large number of convenience features that mainly appeal to the system developer or systems house. Included in a package of this type you should expect to find at least a macro assembler that recognizes several mnemonic dialects and a linkage editor to be used in combining separately assembled subroutines. You may also find some useful utilities such as a cross-reference program.

My own experience in finding the right assembly language development package has ranged from downright horror to moderate satisfaction. Although I must qualify that statement by saying that by the time microcomputers first appeared I had already been spoiled by over 10 years of experience with IBM's mainframe macro assembler. Since that time my satisfaction with it has not wavered and I continue to use it as my basis for comparison. Don't let the word "mainframe" unduly influence your thoughts; I am not about to begin comparing apples to oranges.

In the Beginning

To put things into the proper perspective: I was introduced to IBM's macro assembler on a 32K 360/30. For those of you who have a mainframe background, thoughts of this grand old machine will undoubtedly bring back many fond memories. The standard macro assembler included at no charge with this machine required only a 14K partition of memory to run. If more memory was available it would be used, but only this minimal amount was necessary. If the machine you were using was blessed with a full 64K of memory, a 44K variant of the 14K assembler could be used. Both assemblers offered identical features, only the 44K version executed three or four times faster because fewer overlays were used and more memory was available for table space.

Through the years this same assembler has been transported from generation to generation of IBM computer. With the exception of the additional instructions added with each new generation of processor, the assembler as I knew it has been retained. Certainly this longevity speaks highly of the original design and its acceptance by the user community.

Don't be surprised if the way I have described the 360/30 sounds similar to your microcomputer. The 360/30 CPU was almost as powerful as a 4MHz Z80, and the only area in which the mainframe surpassed today's microcomputer was in the power of its I/O system.

I have used a number of different assembly language development packages marketed by both large and small companies. Many of them offered an excellent value for the price, and a few offered some very exciting, unique features. However, none of them have even come close to satisfying my desire for the powerful and flexible macro facility I had become accustomed to. Practically all of the macro assemblers I have looked at use the Microsoft macro format as a starting point. Any additions are then made around that standard. I have nothing against the Microsoft macro format, as far as it was taken. I simply think that it was considered complete long before it had matured.

Nonetheless, I was unable to find anything better than M80 and out of desperation chose it for my assembler. Actually, my choice was not quite as difficult as I make it sound. M80 is provided as part of a utilities package that is included at no extra charge when you purchase other Microsoft language products. Since I couldn't find anything better than M80 and obviously couldn't beat the price, my decision was easy.

Something New

At CP/M 83 I had the pleasure of meeting Steve Russell, the S and R of SLR Systems. Steve was proudly ex-

hibiting his two latest creations, Z80ASM and Z80LNK. Both are included in an assembly language development package that was purported to be six times faster than M80. I am not a foreigner to M80, and I have never been willing to accept at face value what appears to be an outrageous advertisement, so I asked Steve to send me a review package. After some conversation he asked me if I would have any interest in serving as a Beta test site for Z80ASM after he had added a few more features to make it compatible with M80. I couldn't possibly turn down an offer like that and I left the show questioning Steve's conviction that Z80ASM actually executes six times faster than M80. Don't get me wrong; I'm not suggesting that M80 is fast, just that a speed improvement of *two or three times* would have been more believable.

Around the first of the year the UPS truck stopped to drop off an unexpected package. To my surprise it was from SLR Systems. I had totally forgotten about Z80ASM and my conversation with Steve. Opening the package revealed a three-ring binder of documentation and two disks. The first disk contained the assembler, linkage editor, and installation programs. The other disk contained a number of sample source programs to be used in testing the assembler.

Egg on the Face

While furiously thumbing through the documentation to find the instructions for installing the assembler, I began to chuckle about how easy I thought it was going to be to disprove the advertising claims made about Z80ASM. A few minutes later I had finished configuring the assembler for my system and was ready to run my first test. Rather than using one of the sample source programs included with Z80ASM I chose to use one of my largest programs.

In my haste to get started I had not taken time to thoroughly read the documentation; I ended up blindly specifying every runtime option that looked meaningful. With my stopwatch in hand and a smirk on my face I pressed the return key and anxiously waited for something to happen. And happen it did! After only a few seconds a message was displayed on the CRT suggesting that the first pass was now complete and that the second pass through my program was now in progress. My first thought was that something must have gone wrong because I couldn't believe that the assembler had completed the first pass this quickly. I began to feel a little better when the assembler appeared to be running the second pass much more slowly.

After 25 seconds the end-of-job statistics were displayed on my CRT and control was returned to CP/M. After making a note of the elapsed time taken by Z80ASM I ran the same source program through M80. At the conclusion of the M80 assembly I was satisfied with my suspicion that the claims made about Z80ASM were indeed overstated because M80 took only a little over twice as long as Z80ASM. What I didn't realize was that specifying all the runtime options for Z80ASM commands the program to also create an output listing file including a cross-reference and symbol table. Because I had used one of my largest programs for the test, a listing file of over 160K had been created in addition to the normal .SYM and .REL files. I was suddenly a little more serious about running a complete benchmark after realizing that all of this processing had taken place in 25 seconds. To complete the test with fairness I ran the cross-reference program provided with M80 and directed its output to disk. I then added the runtime of the M80 assembly to that of the cross-reference program to get an overall total.

When I compared the two runtime totals I discovered that my suspicions were ill-founded and that SLR Systems had not overstated the performance of its product. Straight out of the box I achieved the claimed sixfold runtime reduction.

Next month I will talk more about how Z80ASM does its job.

Expert Marriage Counseling

I won't be telling you anything new when I point out that some incompatibilities exist between CP/M 2.2 and its successor, CP/M Plus. Dave Cortesi and I have commented on this in past columns, but both of us failed to provide any solution to the problem. Fortunately, Mike Griswold of Fort Worth, Texas, took the bull by the horns and wrote a Resident System Extension (RSX) that translates version 2.2 BIOS calls to those compatible with CP/M Plus.

After keying it in and correcting the errors that I introduced, I attached it to the public domain DU program. I chose this program to test with because I am familiar with its operation and the reason it fails when run under CP/M Plus. As Mike reported, DU now works perfectly, as do several other programs that I had shelved because of compatibility problems.

The full text of Mike's program can be found in the listing on page 23. It is a great piece of code that is an absolute necessity in warding off many of the mysterious conversion pains likely to occur when going from version 2.2 to CP/M Plus.

Editor's note: Interested readers may also wish to consult the article on using RSXs under CP/M Plus, elsewhere in this issue. **DDJ**

(Listing begins on next page)

CP/M Exchange Listing (Text begins on page 20)

```
;
        title   'CP/M 2.2 BIOS RSX'
;
;       18Jan84          By Mike Griswold
;
;       This RSX will provide CP/M 2.2 compatible BIOS support
;       for CP/M 3.x.  Primarily it performs logical sector
;       blocking and deblocking needed for some programs.
;       All actual I/O is done by the CP/M 3.0 BIOS.
;
        maclib  z80             ; Z80 opcode equates
        cseg
;
;       This equate is the only hardware dependent value.
;       It should be set to the largest sector size that
;       will be used.
;
max$sector$size:        equ     1024
;
;
;       RSX prefix structure
;
        db      0,0,0,0,0,0
entry:  jmp     boot
next:   db      jmp             ; jump
        dw      0               ; next module in line
prev:   dw      0               ; previous module
remove: db      0ffh            ; remove flag
nonbnk: db      0
        db      'BIOS2.21'
        db      0,0,0
;
;       Align jump table on next page boundary.  This is needed
;       for programs that cheat when getting the addresses of
;       BIOS jump table entries.  Optimization freaks could move
;       some code up here.  With a 60K TPA though its hard to
;       get excited.
;
        ds      229
;
;       BIOS Jump Table
;
cbt:    jmp     wboot           ; cold boot entry
wbt:    jmp     wboot           ; warm boot entry
        jmp     xconst          ; console status
        jmp     xconin          ; console input
        jmp     xconout         ; console output
        jmp     xlist           ; list output
        jmp     xauxout         ; aux device output
        jmp     xauxin          ; aux device input
        jmp     home            ; home disk head
        jmp     seldsk          ; select drive
        jmp     settrk          ; select track
        jmp     setsec          ; select sector
        jmp     setdma          ; set dma address
        jmp     read            ; read a sector
```

```
        jmp     write           ; write a sector
        jmp     xliststst       ; list status
        jmp     sectran         ; sector translation
;
;       The CP/M 3.0 BIOS jump table is copied here
;       to allow easy access to its routines.  The disk
;       I/O routines are potentially in banked memory
;       so they cannot be called directly.
;
xwboot: jmp     0               ; warm boot
xconst: jmp     0
xconin: jmp     0
xconout:jmp     0
xlist:  jmp     0
xauxout:jmp     0
xauxin: jmp     0
        jmp     0
        jmp     0
        jmp     0
        jmp     0
        jmp     0
        jmp     0
        jmp     0
xliststst:jmp   0
;
;       Signon message
;
signon: db      0dh,0ah,'BIOS ver 2.21 ACTIVE',0dh,0ah,0
;
;       Cold boot
;
boot:   push    psw             ; a BDOS call is in progress
        push    h               ; so save CPU state
        push    d
        push    b
        lxi     h,next          ; now bypass this RSX on
        shld    entry+1         ; all subsequent BDOS calls
        call    init            ; initialize BIOS variables
        lhld    1               ; save the CP/M 3.0 BIOS jump
        shld    old$addr        ; at location 0
        lxi     d,xwboot        ; set up to move jump table
        lxi     b,15*3          ; byte count
        ldir
        lxi     h,wbt           ; substitute new jump address
        shld    1
        lxi     h,signon        ; sound off
        call    prmsg
        pop     b               ; restore BDOS call state
        pop     d
        pop     h
        pop     psw
        jmp     next            ; carry on
;
;       Warm boot
;
wboot:  lhld    old$addr
```

(Continued on next page)

```
        shld    1               ; restore normal BIOS address
        jmp     0               ; jump to CP/M 3.0 warm boot
;
;       Initialize BIOS internal variables for cold boot
;
init:   xra     a
        sta     hstwrt          ; host buffer written
        sta     hstact          ; host buffer inactive
        lxi     h,80h
        shld    dmaadr
        ret
;
;       Routine to call banked BIOS routines via BDOS
;       function 50.  All disk I/O calls are made through
;       here.
;
xbios:  sta     biospb          ; set BIOS function
        mvi     c,50            ; direct BIOS call function
        lxi     d,biospb        ; BIOS parameter block
        jmp     next            ; jump to BDOS
;
biospb: db      0               ; BIOS function
areg:   db      0               ; A register
bcreg:  dw      0               ; BC register
dereg:  dw      0               ; DE register
hlreg:  dw      0               ; HL register
;
;       Home disk.
;
home:   lda     hstwrt          ; check if pending write
        ora     a
        cnz     writehst        ; dump buffer to disk
        xra     a
        sta     hstwrt          ; buffer written
        sta     hstact          ; buffer inactive
        sta     unacnt          ; zero alloc count
        sta     sektrk          ; zero track count
        sta     sektrk+1
        ret
;
;       Select disk.  Create a fake DPH for programs
;       that might use it.
;
seldsk: mov     a,c             ; requested drive number
        sta     sekdsk
        sta     bcreg           ; set C reg in BIOSPB
        mvi     a,9             ; BIOS function number
        call    xbios           ; CP/M 3.0 select
        mov     a,h
        ora     l               ; check for HL=0
        rz                      ; select error
        mov     e,m             ; get address of xlat table
        inx     h
        mov     d,m
        xchg
```

```
        shld    xlat            ; save xlat address
        lxi     h,11            ; offset to dpb address
        dad     d
        mov     e,m             ; fetch address of dpb
        inx     h
        mov     d,m
        xchg
        shld    dpb             ; address of dpb
        mov     a,m             ; cpm sectors per track
        sta     spt
        inx     h
        inx     h               ; point to block shift mask
        inx     h
        mov     a,m
        sta     bsm             ; save block shift mask
        lxi     d,12            ; offset to psh
        dad     d
        mov     a,m
        sta     psh             ; save physical shift factor
        lxi     h,dph           ; return DPH address
        ret
;
;       This fake DPH holds the addresses of the actual
;       DPB.  The CP/M 3.0 DPH is *not* understood
;       by CP/M 2.2 programs.
;
dph:    equ     $
        dw      0               ; no translation
        ds      6               ; scratch words
        ds      2               ; directory buffer
dpb:    ds      2               ; DPB
        ds      2               ; CSV
        ds      2               ; ALV
;
;       Set track.
;
settrk: sbcd    sektrk
        ret
;
;       Set dma.
;
setdma: sbcd    dmaadr
        ret
;
;       Translate sectors.  Sectors are not translated yet.
;       Wait until we know the physical sector number.
;       This works fine as long as the program trusts
;       the BIOS to do the translation.  Some programs
;       access the XLAT table directly to do their own
;       translation.  These programs will get the wrong
;       idea about the disk skew but it should cause no
;       harm.
;
sectran:mov     l,c             ; return sector in HL
        mov     h,b
        ret
;
;       Set sector number.
;
setsec: mov     a,c
```

(Continued on page 30)

CP/M Exchange Listing (Listing Continued, text begins on page 20)

```
        sta     seksec
        ret
;
;       Read the selected CP/M sector.
;
read:   mvi     a,1
        sta     readop          ; read operation
        inr     a               ; a=2 (wrual)
        sta     wrtype          ; treat as unalloc
        jmp     alloc           ; perform read
;
;       Write the selected CP/M sector.
;
write:  xra     a
        sta     readop          ; not a read operation
        mov     a,c
        sta     wrtype          ; save write type
        cpi     2               ; unalloc block?
        jrnz    chkuna
;
;       Write to first sector of unallocated block.
;
        lda     bsm             ; get block shift mask
        inr     a               ; adjust value
        sta     unacnt          ; unalloc record count
        lda     sekdsk          ; set up values for
        sta     unadsk          ; writing to an unallocated
        lda     sektrk          ; block
        sta     unatrk
        lda     seksec
        sta     unasec
;
chkuna: lda     unacnt          ; any unalloc sectors
        ora     a               ; in this block?
        jrz     alloc           ; skip if not
        dcr     a               ; unacnt=unacnt-1
        sta     unacnt
        lda     sekdsk
        lxi     h,unadsk
        cmp     m               ; sekdsk = unadsk ?
        jrnz    alloc           ; skip if not
        lda     sektrk
        lxi     h,unatrk
        cmp     m               ; sektrk = unatrk ?
        jrnz    alloc           ; skip if not
        lda     seksec
        lxi     h,unasec
        cmp     m               ; seksec = unasec ?
        jrnz    alloc           ; skip if not
        inr     m               ; move to next sector
        mov     a,m
        lxi     h,spt           ; addr of spt
        cmp     m               ; sector > spt ?
        jrc     noovf           ; skip if no overflow
        lhld    unatrk
        inx     h
```

```
        shld    unatrk          ; bump track
        xra     a
        sta     unasec          ; reset sector count
noovf:  xra     a
        sta     rsflag          ; don't pre-read
        jr      rwoper          ; perform write
;
alloc:  xra     a               ; requires pre-read
        sta     unacnt
        inr     a
        sta     rsflag          ; force pre-read
;
rwoper: xra     a
        sta     erflag          ; no errors yet
        lda     psh             ; get physical shift factor
        ora     a               ; set flags
        mov     b,a
        lda     seksec          ; logical sector
        lxi     h,hstbuf        ; addr of buffer
        lxi     d,128
        jrz     noblk           ; no blocking
        xchg                    ; shuffle registers
shift:  xchg
        rrc
        jrnc    sh1
        dad     d               ; bump buffer address
sh1:    xchg
        dad     h               ; double offset
        ani     07fh            ; zero high bit
        djnz    shift
        xchg                    ; HL=buffer addr
noblk:  sta     sekhst
        shld    sekbuf
        lxi     h,hstact        ; buffer active flag
        mov     a,m
        mvi     m,1             ; set buffer active
        ora     a               ; was it already?
        jrz     filhst          ; fill buffer if not
        lda     sekdsk
        lxi     h,hstdsk        ; same disk ?
        cmp     m
        jrnz    nomatch
        lda     sektrk
        lxi     h,hsttrk        ; same track ?
        cmp     m
        jrnz    nomatch
        lda     sekhst          ; same buffer ?
        lxi     h,hstsec
        cmp     m
        jrz     match
;
nomatch:
        lda     hstwrt          ; buffer changed?
        ora     a
        cnz     writehst        ; clear buffer

filhst: lda     sekdsk
        sta     hstdsk
        lhld    sektrk
```

(Continued on next page)

CP/M Exchange Listing (Listing Continued, text begins on page 20)

```
        shld    hsttrk
        lda     sekhst
        sta     hstsec
        lda     rsflag          ; need to read ?
        ora     a
        cnz     readhst         ; yes
        xra     a
        sta     hstwrt          ; no pending write
;
match:  lhld    dmaadr
        xchg
        lhld    sekbuf
        lda     readop          ; which way to move ?
        ora     a
        jrnz    rwmove          ; skip if read
        mvi     a,1
        sta     hstwrt          ; mark buffer changed
        xchg                    ; hl=dma  de=buffer
;
rwmove: lxi     b,128           ; byte count
        ldir                    ; block move
        lda     wrtype          ; write type
        cpi     1               ; to directory ?
        jrnz    exit            ; done
        lda     erflag          ; check for errors
        ora     a
        jrnz    exit            ; don't write dir if so
        xra     a
        sta     hstwrt          ; show buffer written
        call    writehst        ; write buffer
exit:   lda     erflag
        ret
;
;       Disk read.  Call CP/M 3.0 BIOS to fill the buffer
;       with one physical sector.
;
readhst:
        call    rw$init         ; init CP/M 3.0 BIOS
        mvi     a,13            ; read function number
        call    xbios           ; read sector
        sta     erflag
        ret
;
;       Disk write.  Call CP/M 3.0 BIOS to write one
;       physical sector from buffer.
;
writehst:
call    rw$init         ; init CP/M 3.0 BIOS
mvi     a,14            ; write function number
call    xbios           ; write sector
sta     erflag
ret
```

```
;
;       Translate sector.  Set CP/M 3.0 track, sector,
;       DMA buffer and DMA bank.
;
rw$init:
        lda     hstsec          ; physical sector number
        mov     l,a
        mvi     h,0
        shld    bcreg           ; sector number in BC
        lhld    xlat            ; address of xlat table
        shld    dereg           ; xlat address in DE
        mvi     a,16            ; sectrn function number
        call    xbios           ; get skewed sector number
        mov     a,l
        sta     actsec          ; actual sector
        shld    bcreg           ; sector number in BC
        mvi     a,11            ; setsec function number
        call    xbios           ; set CP/M 3.0 sector
        lhld    hsttrk          ; physical track number
        shld    bcreg           ; track number in BC
        mvi     a,10            ; settrk function number
        call    xbios
        lxi     h,hstbuf        ; sector buffer
        shld    bcreg           ; buffer address in BC
        mvi     a,12            ; setdma function number
        call    xbios
        mvi     a,1             ; DMA bank number
        sta     areg            ; bank number in A
        mvi     a,28            ; setbnk function number
        call    xbios           ; set DMA bank
        ret
;
;       Print message at HL until null.
;
prmsg:  mov     a,m
        ora     a
        rz
        mov     c,m
        push    h
        call    xconout
        pop     h
        inx     h
        jmp     prmsg
;
;       disk i/o buffer
;
hstbuf: ds      max$sector$size
;
;       variable storage area
;
sekdsk: ds      1               ; logical disk number
```

(Continued on next page)

CP/M Exchange Listing (Listing Continued, text begins on page 20)

```
sektrk: ds      2               ; logical track number
seksec: ds      1               ; logical sector number
;
hstdsk: ds      1               ; physical disk number
hsttrk: ds      2               ; physical track number
hstsec: ds      1               ; physical sector number
;
actsec: ds      1               ; skewed physical sector
sekhst: ds      1               ; temp physical sector
hstact: ds      1               ; buffer active flag
hstwrt: ds      1               ; buffer changed flag
;
unacnt: ds      1               ; unallocated sector count
unadsk: ds      1               ; unalloc disk number
unatrk: ds      2               ; unalloc track number
unasec: ds      1               ; unalloc sector number
sekbuf: ds      2               ; logical sector address in buffer
;
spt:    ds      1               ; cpm sectors per track
xlat:   ds      2               ; xlat address
bsm:    ds      1               ; block shift mask
psh:    ds      1               ; physical shift factor
;
erflag: ds      1               ; error reporting
rsflag: ds      1               ; force sector read
readop: ds      1               ; 1 if read operation
rwflag: ds      1               ; physical read flag
wrtype: ds      1               ; write operation type
dmaadr: ds      2               ; last dma address
oldaddr:ds      2               ; address of old BIOS
;
        end
```

End Listing

Resident System Extensions

RSX Under CP/M Plus

One of the many new features of CP/M Plus is the ability to use a Resident System Extension (RSX). An RSX allows a programmer to extend or modify any BDOS function, as well as create new ones. To access the BDOS under CP/M, a program loads certain registers with parameters and makes a call to location 0005h. Stored at this location is a jump to the beginning of BDOS. After an RSX has been loaded, the jump instruction points instead to the beginning of the RSX. This allows the RSX to intercept all BDOS calls and to modify them as the programmer sees fit. One or more RSXs can be attached to an executable file using the CP/M Plus system utility GENCOM; they are loaded at the same time as the program.

The applications of an RSX may at first seem limited, but after you have written your first one the uses will seem endless. The idea for my first RSX came about one day while I was playing Pacman on my computer. Every time the little guys moved, the terminal beeped, and there was no way to pause the program if I were interrupted. The answer was simple: *I* wrote a 37-line RSX that intercepted all console output calls (BDOS functions 2 and 6) and all console input calls (functions 1 and 6).

When the program outputs a character, the RSX compares it with the bell character. If it is a bell, the RSX simply returns to the program; otherwise, it passes the call on to BDOS. When the program makes a console input call, the RSX makes a corresponding console input call and compares the character returned from BDOS with the ESC character. If it is not the ESC character, the RSX returns the character to the calling program. If it is an ESC, the RSX does a console input call using BDOS function 6 (Direct Console I/O) with register E set to 0FDh. This suspends the calling process until a character is typed. Therefore, when you press the ESC key while the program is running, the program is suspended until you press another key.

You could attach this RSX to any other program where you need the ability to temporarily pause the program but it is not available. You could also modify this RSX to redefine any set of keys on the keyboard. RSXs can become much more complex, including routines that monitor a modem port and notify you when someone is trying to access your computer and complete disk or character I/O drivers.

by Garry M. Silvey

Garry Silvey, Digital Research, P.O. Box 579, 160 Central Avenue, Pacific Grove, CA 93950.

CP/M Plus is a registered trademark of Digital Research, Inc. LINK and RMAC are trademarks of Digital Research, Inc.

RSX Memory Organization

RSXs are loaded when a program containing an RSX is loaded. When the loader detects that the file being loaded contains an RSX, it strips the RSX from the program code, relocates the RSX just below BDOS (or a previously loaded RSX), modifies the jump instruction at location 0005h to point to the beginning of the RSX, and loads the program code beginning at location 0100h in memory. The loader also sets the uninitialized fields in the RSX Prefix.

The figure (page 37) illustrates how memory is configured before and after an RSX has been loaded. Note that the size of the transient program area will shrink as each new RSX is loaded.

If more than one RSX exists in memory, the jump instruction at location 0005h points to the first RSX. The Next field of that RSX points to the next RSX, and so on until the chain of RSXs reaches BDOS.

Format of an RSX

The first 27 bytes of an RSX contain a data structure called the RSX Prefix that is used by both the RSX and the loader. In the RSX Prefix is a flag that indicates whether the RSX should be loaded in nonbanked systems only and whether or not the RSX should be removed during a warm boot. If the RSX is not removed at warm boot, it will be active until the system is cold-booted; all programs that are executed after the RSX has been loaded will have their BDOS calls intercepted by the RSX.

An example of an RSX that loads only in nonbanked systems is the DIRLBL.RSX file that is included with CP/M Plus. This RSX, attached to SET.COM, modifies the way the directory label is updated in nonbanked systems. If SET is run on a banked system, the RSX is not loaded. The table (page 37) contains a listing of all of the fields contained in the RSX Prefix. Listing One (page 40) shows how an RSX Prefix would be set up in assembly language.

The fields that the programmer must initialize are the Start, Remove, Nonbanked, and Name fields, as well as part of the Next field. The loader will fill in the other fields when it loads the RSX into memory.

The first byte of the Next field should be initialized to 0c3h, the hexadecimal code for the jump instruction. The Start field should be initialized with a jump instruction to the beginning of the RSX code. This is where a function number is tested to see if the RSX should intercept it.

The Remove flag should be set to 0ffh if the RSX is to be removed from memory by the next call to BDOS function 59, Load Overlay. If this flag is set to zero, the RSX will remain in memory and be functional until the system is cold-booted.

Note that the CCP always calls the Load Overlay function during a warm boot.

The Nonbanked flag is set to 0ffh to tell the loader to load the RSX in nonbanked systems only. If the flag is set to zero, the RSX is loaded in both banked and nonbanked systems. The Name field should be initialized to the name of the RSX; the name must be 8 bytes long, so pad it with blanks if necessary.

Function of an RSX

Once a program that contains an RSX has been loaded, the RSX intercepts all BDOS calls (Call 0005h) before they reach BDOS. The RSX then determines if the function number being passed to BDOS matches a function number that it should extend, modify, or create. If the function is not relevant to the RSX, the call is passed up the RSX chain by jumping to the Next field of the RSX Prefix.

If the RSX needs to make a BDOS call, it must call the address stored in the Next field instead of the address at location 0005h. By using the address in the Next field, the call starts with the next RSX or BDOS. If the RSX makes a BDOS call using location 0005h, the call will eventually reach the same RSX, which could result in an infinite loop!

The actions that an RSX can take are many. The RSX may simply modify the parameters passed to it, then let the call continue on to BDOS; in this case, the last instruction of the RSX would be a JMP NEXT. The RSX may catch the call, perform a number of BDOS calls using the instruction CALL NEXT, and then either return to the transient program by using a RET instruction or pass the original call on to BDOS by using a JMP NEXT instruction. The RSX may also catch a BDOS call, perform the BDOS action (i.e., direct I/O port accessing) without ever using BDOS, and then return to the calling program.

The RSX should set up a local stack that is large enough for its own needs. If it does set up its own stack, it must restore the stack pointer to the entry stack before exiting.

RSX Example

The RSX example included with this article will kill two birds with one stone. It makes a good example for how to use RSX, and it "patches" an incompatibility between CP/M 2.2 and CP/M Plus.

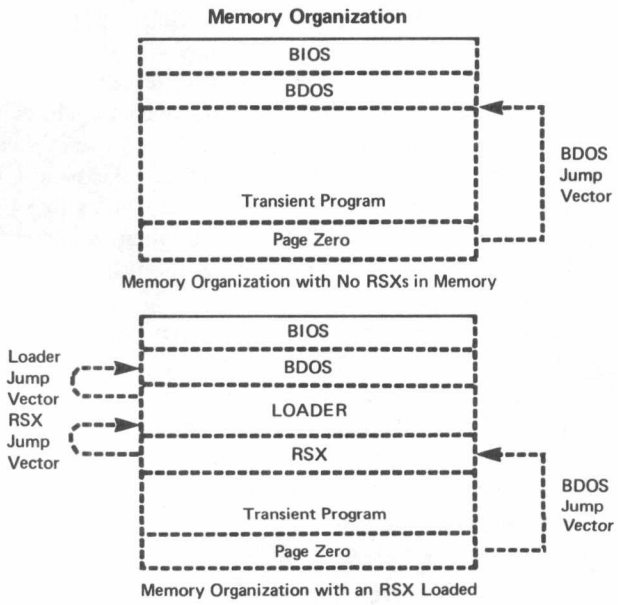
In CP/M 2.2 the filename specified in the FCB given for BDOS function 15, Open File, can contain question marks; the BDOS will open the first file whose name matches the ambiguous filename. If the same program is run under CP/M Plus, the BDOS error "? in filename" occurs and an error is returned. If a program you have developed under 2.2 calls the Open File function with question marks in the FCB, you simple attach the following RSX to it, and it will run under CP/M Plus as expected.

The sample RSX modifies the BDOS Open File function with the following routine (see Listing Two, page 40). When BDOS is called with the function number equal to 15, the RSX intercepts the call and checks the FCB for question marks. If no question marks appear in the FCB, the RSX passes the call on to BDOS for normal processing. If there are question marks in the FCB, the RSX uses the ambiguous filename in the FCB in a call to BDOS function 17, Search

Serial Number:	*Holds the serial number of the operating system.*
Start*:	*Contains a jump instruction to the beginning of the* RSX *code.*
Next:	*Contains a jump instruction to the next RSX in the* RSX *chain or to* BDOS.
Previous:	*Contains the address of the previous (newer)* RSX *or location 5 if this is the first* RSX.
Remove*:	*This flag tells the loader whether or not the* RSX *should be removed during the next call to the loader via* BDOS *function 59, Load Overlay. (This function is always called during warm boot.)*
Nonbanked*:	*Tells the loader if the* RSX *should be loaded in nonbanked systems only.*
Name*:	*Contains an 8-character name for the* RSX.
Loader:	*This flag is set if the* RSX *is the last one in the* RSX *chain. If the flag is set, it indicates that the* RSX *is actually the loader.*
Reserved:	*This field is reserved for use by* BDOS.

**The programmer must initialize this field.*

Table
RSX Prefix Fields



Figure

First. If the search returns an error, the RSX returns a "file not found" error to the program. If the search is successful, the RSX takes the filename found and places it into the user's FCB. It then jumps to BDOS with register C set to the Open File function. BDOS then opens the file as expected and returns to the program.

Also included with this article is a short program that tests the functionality of the RSX example (Listing Three, page 43). It prompts the user for a filename, uses BDOS function 152, Parse Filename, to set up an FCB, then calls BDOS function 15, Open File. If the RSX has been attached, and the user gives an ambiguous filename when prompted, the first file whose name matches the filename is opened as it would under CP/M 2.2. If the RSX has not been attached, an error is returned.

To use the RSX in Listing Two, first type it in using your editor, then perform the following steps:

(1) Use RMAC to assemble the file.

(2) Use LINK to link the file. Use the OP option so LINK will produce a Page Relocatable (.PRL) file.

(3) RENAME the file created by LINK so that it will have a filetype of .RSX.

(4) Use the test program supplied, or a program that you know uses FCBs with question marks, and attach the RSX file to it using GENCOM. **DDJ**

(Listings begin on page 40)

Example

```
13A>RMAC OPENRSX
CP/M RMAC ASSEM 1.1
00C7
001H USE FACTOR
END OF ASSEMBLY

13A>LINK OPENRSX [OP]
LINK 1.31
TRUE    FFFF    FALSE   0000    PRINTS  0009    OPENFI  000F
SEARCH  0011    SCB     0031    CALLRS  003C    DEBUG   FFFF
NEXT    0109    PREV    010C    REMOVE  010E    NONBNK  010F
RSXNAM  0110    START   011B    CAUGHT  0132    LOOP    0141
DOIT    0155    FOUND   0170    LDIR    0189    SCBPB   019A
OFFSET  019A    OPERAT  019B    VALUE   019C    USERST  019E
RSXSTA  01AA    QMESS   01AA

ABSOLUTE            0000
CODE SIZE   00C7    (0100-01C6)
DATA SIZE           0000
COMMON SIZE         0000
USE FACTOR          04

13A>REN OPENRSX.RSX=OPENRSX.PRL
13A>MAC OPEN
CP/M MACRO ASSEM 2.0
0233
001H USE FACTOR
END OF ASSEMBLY

13A>HEXCOM OPEN
HEXCOM VERS: 3.00

FIRST ADDRESS       0100
LAST ADDRESS        0232
BYTES READ          0101
RECORDS WRITTEN     03

13A>GENCOM OPEN OPENRSX

GENCOM completed.

13A>
```

The RSX assembly language file (Listing One) is called OPENRSX.ASM, and the test program (Listing Three) is called OPEN.ASM.

RSX Under CP/M (Text begins on page 36)

Listing One

RSX Prefix Assembly Language Example

```
serial:    db    0,0,0,0,0,0     ; The loader will put the
                                 ;    serial number here.
start:     jmp   RSXstart        ; Jump to start of the
                                 ;    RSX code.
next:      jmp   $-$             ; jmp instruction
                                 ; Address of the next
                                 ;    module, this is
                                 ;    filled in by the
                                 ;    loader.
prev:      dw    0               ; The loader will put the
                                 ;    address of the
                                 ;    previous module here
remove:    db    0ffh            ; Remove flag
nonbank:   db    0               ; Nonbank flag
name:      db    'RSX NAME'      ; Name of the RSX
Loader:    db    0               ; Loader flag
           db    0,0             ; Reserved for system use

RSXstart:                        ; Actual RSX code starts here
```

End Listing One

Listing Two

Open File RSX

```
;
; This program is assembled using RMAC and LINK to create an RSX to
; catch BDOS open file calls which specify an FCB that has question
; marks in it.  If there are no ?'s in the FCB, the call is passed on
; to BDOS.  If there are ?'s, this RSX will use the FCB pointed to by
; register pair DE to call the BDOS search first function.  If the
; search is not successful, it will return the appropriate error to
; the calling program.  If the search was successfull, the filename
; found is transfered from the DMA buffer to the FCB address that was
; originally passed in register DE.  The RSX will then execute a normal
; open file call.
;

TRUE    equ     0ffffh
FALSE   equ     NOT TRUE

PRINTSTR        equ     09              ; Print String
OPENFILE        equ     15              ; Open File
SEARCHF         equ     17              ; Search for First
SCB             equ     49              ; Get/Set System Control Block
CALLRSX         equ     60              ; Call RSX

DEBUG   equ     TRUE                    ; will display more messages
                                        ; while debuging
```

```
; Start of the RSX Prefix
        db      0,0,0,0,0,0             ; serial number
        jmp     START                   ; start of this RSX
NEXT    jmp     $-$                     ; jmp instruction
PREV    dw      0                       ; address of previous module
REMOVE  db      0ffh                    ; remove on warm boot
NONBNK  db      0                       ; load for banked and non-banked
RSXNAME db      'OPEN    '              ; name of RSX
        db      0,0,0                   ; used by BDOS

; start of RSX code
START:  mov     a,c                     ; check BDOS function number
        cpi     OPENFILE                ; is it an open file call ?
        jz      CAUGHT                  ; if so, process it
; If DEBUG is true, then check the function number for the BDOS function
; Call RSX.  If the function number matches, return to the calling program
; with register A set to 0 to tell the program that the RSX has been loaded.
if DEBUG
        cpi     CALLRSX                 ; is it a call to this RSX ?
        jnz     NEXT                    ; nope, pass to next RSX or BDOS
        mov     a,d                     ; Is it the test program calling ?
        cpi     0ffh
        jnz     NEXT
        cmp     e                       ; register DE equal 0ffffh ?
        jnz     NEXT                    ; nope, pass call on to next module
        xra     a                       ; otherwise zero A
        ret                             ; return to calling program
else
        jmp     NEXT                    ; jump to next module if not open call
endif

CAUGHT: lxi     h,0
        dad     sp                      ; get current stack pointer
        shld    USERSTACK               ;       and save it
        lxi     sp,RSXSTACK             ; use local stack
        push    b                       ; save users registers
        push    d

;       check filename for ?'s
        mvi     b,11                    ; max # of characters in a filename
        inx     d                       ; point to filename in FCB
LOOP:   ldax    d
        cpi     '?'                     ; is it a ?
        jz      DOIT                    ; found a ?, so execute rest of RSX
        inx     d                       ; try next character
        dcr     b                       ; decrement counter
        jnz     LOOP

;       If the program has reached here, there are no ?'s in the filename
;       so pass the call on to BDOS (or the next RSX) for normal execution.

        pop     d                       ; restore the users registers
        pop     b
        lhld    USERSTACK
        sphl                            ; restore users stack
        jmp     NEXT                    ; pass to next RSX or BDOS

DOIT:   ; Here a Search First function is executed, then the FCB is
        ; updated with the filename that was found, and then a Open
        ; File command is sent to BDOS.  If the search first call
        ; returns an error, the RSX returns the appropriate error.

if DEBUG
        ; Tell the user we found a "?"
        lxi     d,QMESS
```

(Continued on next page)

Listing Two

```
        mvi     c,PRINTSTR
        call    NEXT
endif
        pop     d                       ; get the FCB address
        push    d                       ; save it again
        mvi     c,SEARCHF               ; prepare for a Search First call
        call    NEXT                    ; (call BDOS)
        cpi     0ffh                    ; not found ?
        jnz     FOUND                   ; continue processing
        ; An error occured during the Search First call. In this
        ; case, we do not want to pass the original call onto BDOS since
        ; there are still question marks in the FCB.  Instead, return to
        ; the calling program with errors in a,hl as set by BDOS.
        pop     d                       ; restore users registers
        pop     b
        lhld    USERSTACK
        sphl                            ; restore users stack
        ret                             ; return to users program

FOUND:  ; The Search First was successfull, so update the users FCB
        ; with the FCB of the file that was found, and jump to BDOS's
        ; Open File function.

        ; First multiply directory code by 32 to get offset to the
        ; FCB of the file that was found.
        add a  !  add a
        add a  !  add a
        add a
        push    a                       ; save offset

        ; The value of the current DMA address must be determined so
        ; the RSX can retrieve the FCB of the file that was just found.
        ; The calling program may have modified the current DMA address,
        ; so the DMA address will have to be extracted from the SCB.

        mvi     c,SCB                   ; get/set SCB function
        lxi     d,SCBPB                 ; SCB pb address
        call    NEXT                    ; call BDOS
;       Register pair HL now contain the DMA address
        pop     a                       ; restore the offset to the FCB
        mov     c,a
        mvi     b,0                     ; put offset into bc
        dad     b                       ; hl now points to the FCB of the file
                                        ; that was just found
        inx     h                       ; point hl to start of file name
        pop     d                       ; get d (which points to the users FCB)
        push    d                       ; save d again
        inx     d                       ; point to file name
        ; Now the calling program's FCB must be updated with the FCB of
        ; the filename that was just found.  The filename field of the
        ; FCB is the only part that needs to be updated.
        ;
        ; If you have a Z80 CPU, use the following code.
;       lxi     b,11
;       dw      0b0edh                  ; Z80 ldir instruction
;
        ; If you have a 8080 CPU, use the following code.
        mvi     b,11
LDIR:   mov     a,m ! inx h
        stax    d   ! inx d
```

```
        dcr     b
        jnz     LDIR

        ; The calling program's FCB now has a full filename in it, so
        ; now the RSX can setup an Open File call.
        pop     d
        pop     b                       ; Restore users registers.  Register
                                        ; C will be restored with the
                                        ; original Open File function
                                        ; number in it.
        lhld    USERSTACK
        sphl                            ; restore users stack
        jmp     NEXT                    ; BDOS will now open the file normally

SCBPB:  ; SCB parameter block: Set up to Get the DMA address stored
        ; in the SCB at offset 03ch.
OFFSET:         db      03ch                    ; offset in SCB to DMA address
OPERATION:      db      0                       ; get operation
VALUE:          dw      0

USERSTACK:      dw      0                       ; storage for users stack

                ds      10                      ; space for stack
RSXSTACK:

        ; If debug is true, include the following message.
if DEBUG
QMESS   db      '? in FCB, caught by RSX...',0dh,0ah,'$'
endif
```

(End Listing Two)

Listing Three

Open File RSX Test Program

```
;
; Sample program to test the RSX in listing #2.  The program asks
; for a filename and builds and FCB for the file using the BDOS
; Parse Filename function.  It then tries to open the file.
;
; If the filename has any question marks in it, BDOS would normally
; terminate the program and display an error message.  After the RSX
; is installed, the file is opened as under CP/M 2.2 and no error is
; returned.
;

; miscellaneous equates
TRUE    equ     0ffffh
FALSE   equ     not TRUE
BDOS    equ     5                       ; BDOS jump address
CR      equ     13                      ; carriage return
LF      equ     10                      ; line feed

PRINTSTR        equ     09              ; Print String
RBUFF           equ     10              ; Read Console Buffer
OPENFILE        equ     15              ; Open File
SEARCHF         equ     17              ; Search for First
SCB             equ     49              ; Get/Set SCB
CALLRSX         equ     60              ; Call RSX
```

(Continued on next page)

Listing Three

```
PARSE           equ     152             ; Parse Filename

DEBUG   equ     TRUE

        org     0100h

START:  ; print signon message
        lxi     d,SIGNON
        mvi     c,PRINTSTR
        call    BDOS

if DEBUG
        ; See if the RSX has been attached...
        mvi     c,CALLRSX
        ; Normally, register pair DE would point to a RSX parameter
        ; block, but since the RSX will not use any parameters, load
        ; DE with a value of 0ffffh to differentiate this RSX call from
        ; other RSX calls that could be used by other programs.
        lxi     d,0ffffh
        call    BDOS
        cpi     0                       ; is there a RSX attached ???
        jz      PRSXM                   ; yes, tell user
        mvi     c,PRINTSTR
        lxi     d,NO                    ; tell the user there is no RSX
        call    BDOS
        ; Print RSX message
PRSXM   mvi     c,PRINTSTR
        lxi     d,RSXM
        call    BDOS
endif

        ; Prompt user for filename
        mvi     c,PRINTSTR
        lxi     d,PROMPT
        call    BDOS

        ; Get the filename from the user
        lxi     d,INBUFF
        mvi     c,RBUFF
        call    BDOS

        ; Move the cursor to the next line
        lxi     d,RETURN
        mvi     c,PRINTSTR
        call    BDOS

        ; Set up the Parse Filename Control Block and parse the filename
        lxi     d,PFCB
        mvi     c,PARSE                 ; BDOS parse file name function
        call    BDOS

        ; The parsed filename is now in the FCB, so now open it
        lxi     d,FCB
        mvi     c,OPENFILE              ; BDOS open file function
        call    BDOS
        cpi     0ffh                    ; error returned ?
        jnz     FOUND                   ; if not exit program
```

```
        ; Print an error message if the file was not found
        lxi     d,ERRMESS               ; print error message
        mvi     c,PRINTSTR              ; BDOS print string function
        call    BDOS
EXIT:   ; Return control back to the CCP
        mvi     c,0                     ; BDOS warm boot function
        jmp     BDOS

FOUND:  ; Tell the user what the name of the file that was found is
        lxi     d,ENDMESS
        mvi     c,PRINTSTR
        call    BDOS                    ; print ending message
        lxi     d,FCB+12                ; point de to end of filename
        mvi     a,'$'
        stax    d                       ; put a $ there
        lxi     d,FCB+1                 ; point d to first character of name
        mvi     c,PRINTSTR
        call    BDOS                    ; print the filename
        lxi     d,RETURN                ; drop to next line
        mvi     c,PRINTSTR
        call    BDOS
        jmp     EXIT                    ; return to ccp

PFCB:   ; Parse Filename Control Block
        dw      INBUFF+2                ; address of input string
        dw      FCB                     ; address of target FCB
INBUFF: ; Console input buffer
        db      14,0                    ; number of characters to read
        ds      14                      ; input buffer space
FCB:    ds      36                      ; FCB location

; Terminal messages
ERRMESS:db      'file not found',cr,lf,'$'
SIGNON: db      'open file RSX test program...',cr,lf,'$'
ENDMESS:db      'name of file found : $'
RETURN: db      cr,lf,'$'
RSXM:   db      'RSX has been installed.',cr,lf,'$'
PROMPT: db      'enter file name : $'
NO:     db      'no $'
        end
```

End Listings

p–A Small C Preprocessor

Last month Dr. Schreiner provided us with cc, *a driver for a Small C programming system. This month he presents* p, *the preprocessor for Small C referred to in the previous article.* p *provides yet another tool for greater convenience and power in the Small C environment. Also included this month is code for adding random order release of memory to the routine library. While not included with* cc *last month, it is applicable in that context as well.—Ed.*

Jim Hendrix's Small C compiler (*DDJ*, December 1982) supports some of the preprocessor commands that are usually available in C systems: a symbolic name can be defined for an arbitrary text, which will then be inserted whenever the name appears in the source program. This facility is most frequently used to give meaningful names to various important constants in a program, but it can also be used to give C programs the appearance (almost) of Pascal programs, to substitute one function name for another, and so on. Hendrix's compiler also supports one level of file inclusion, although with a somewhat nonstandard syntax, and it supports conditional compilation based on whether or not certain symbolic names have been defined.

Such a preprocessor is an important tool in its own right. It can be combined with other language processors and assemblers as well. It also becomes considerably more flexible if text substitution can be parameterized (i.e., if macro calls can have arguments and if file inclusion is performed to arbitrary depth). If the preprocessor eliminates C-style comments and translates C-style constants to decimal notation, it simplifies the compiler's job significantly and at the same time obtains a uniform approach to constant notation, commenting conventions, and file inclusion.

p, the program described in this article (and presented in the Listing One on page 52), is such an independent preprocessor. It is written in Small C and supports all the features of the regular C preprocessor described by Kernighan and Ritchie (*The C Programming Language*, Prentice Hall, 1978) with the exception of an *if* preprocessor statement with a constant expression argument.

by Axel T. Schreiner

Axel T. Schreiner, Universität Ulm, Sektion Meth.d.Informatik, Oberer Eselsberg, 7900 Ulm (Doran) West Germany.

**Unix is a trademark of Bell Laboratories.*

Features

p is based on a runtime support that passes arguments to the main program; it expects to be called as follows:

p (option) . . . (inputfile (outputfile))

If no files are explicitly specified, *p* will read from "standard input" and write to "standard output" (the runtime support presumably makes those connections as well). If the runtime support would normally connect to the terminal, it is quite simple to test certain features through input from the terminal or to see the results of a preprocessor run directly at the terminal.

You may specify options in any order. They are generally cumulative. The following options are available:

-d name (=value)	Define a symbolic name
-e	Suppress position stamps
-i drive	Search for file inclusion
-u name	Undefine a symbolic name
-v	Verbose — for debugging

You may redefine symbolic names when *p* is called. This is quite convenient for maintaining various versions of a program. *p* predefines the name cpm. You can undo this (in particular) using the option to undefine a name. To permit a compiler to emit error messages referencing the original source files, *p* will create a position stamp (i.e., a line starting in "#" and containing a decimal line number and, if known, a filename whenever this is necessary for correct sequencing of output lines. Hendrix's Small C compiler cannot handle these position stamps; therefore, they can be suppressed.

CP/M identifies files by name and disk drive. *p* therefore will search *include* files on various disks: first on the disk of the input file (which may be the currently selected disk) and last on the currently selected disk. In between, *p* may search other disk drives optionally, if appropriate options have been specified. The search proceeds from right to left over the *-i* options.

The options are clearly patterned after the Unix* system. The main program expects to receive pointers to the various options as a vector "argv." The number of such options, including (theoretically) the program name as first option, is also passed as an integer "argo." *p* is tolerant enough to accept the parameter of an option (such as an *include* drive) as part of the option or as a separate, immediately following option. Options, however, must precede the filenames.

Once started, *p* will read the input file and write the output file. C-style comments (i.e., arbitrary texts enclosed in /* and */ sequences) are replaced by single blanks. Next preprocessor command lines are processed. Then in regular text lines macro calls (i.e., appearances of defined names possibly followed by a list of argu-

ments in parentheses) are replaced. The replacement text is surrounded by blanks and is reprocessed for further macro calls. Recursion may happen, but reprocessing is aborted after a few attempts with an appropriate message.

Input lines, as well as output lines, may be arbitrarily long. You may continue input lines over several source lines by placing a backslash right before the end of the source line to be continued. C is free format, but preprocessing is line oriented; continuation should be necessary (and come in handy) only for preprocessor command lines, macro calls, and very long strings. Macro calls and strings must be fully contained within one (possibly continued) input line.

A macro call is not recognized within a comment, a preprocessor command line, a string, or a character constant. The replacement text is surrounded by blanks. You cannot use two adjacent macro calls to create another macro call from their replacement text.

If you define a macro with parameters, the macro call consists of the macro name and a list of arguments, separated by commas and enclosed by parentheses. While in the definition the left parenthesis must immediately follow the macro name, it need not in the call. The call, however, must be completely contained on one input line to avoid rather sticky questions should macro be redefined inside its call, etc.

define name replacement-text
define name(name,. . .) replacement-text

undef name

ifdef name
ifndef name
else
endif

include "filename"
include (filename)

line linenumber filename

define is used to define a symbolic name for an arbitrary replacement text. You may parameterize the replacement text. The parameter names are local to each definition, and unfortunately no test exists to determine whether two parameter names within one definition are one and the same. Names follow C conventions; i.e., they must start in a letter or underscore, can contain letters, digits, or underscores, and can be arbitrarily long.

Redefinition is possible but provides a message, *undef* can be used to remove a definition, and there is no complaint if the relevant name was never defined.

ifdef is fulfilled if the specified name is currently defined; *ifndef* is fulfilled if it is not. In either case, subsequent input lines (including preprocessor commands) are processed or skipped depending on the condition. *else* reverses the current value of the condition, and *endif* terminates the construct. You may nest these constructs to any depth. *else* should only be used once per construct, but this is not checked. Each *else* reverses the current state of things.

include causes file inclusion. The filename should follow CP/M conventions. If it does not contain an explicit disk drive specification, the file is searched on the list of drives beginning with the input file drive and ending with the currently selected drive. If the filename is enclosed in angle brackets rather than double quotes, the first drive on the list is skipped. (Normally, brackets are used to designate public *include* files, presumably residing on the currently selected disk and not on the disk of the input file.)

A macro argument is an arbitrary text and may even contain a comma within balanced parentheses, a string, or a character constant. The text is substituted wherever the corresponding parameter appears in the macro definition, even within a string. No blanks are forced around the argument text. There must be exactly as many arguments as there are defined parameters.

p removes leading white space from the output lines and converts constants to decimal notation. These "features" can be removed easily. They do impair the general applicability of the program, but they overcome certain problems in Hendrix's compiler while significantly shortening the output file.

The following constant notations are accepted and are converted to (signed) decimal representation:

digits	decimal constant
0 digits	octal constant(digits 0 . . 7)
0x digits	hexadecimal constant
0X digits	(digits 0 . . 9, a . . f, A . . F)
'c'	character constant

Clearly, no blank may separate the base prefix from the actual constant. Character constants may be escaped; escape sequences consist of a backslash character followed by other characters. The following escapes are recognized:

b	backspace
f	form feed
n	newline (line feed)
r	return
t	tab
'	single quote
"	double quote
\	backslash itself
d	octal, up to three digits 0 . . 7

Character constants are converted to decimal notation. This is reasonable for C programs, but it might cause problems elsewhere.

Preprocessor command lines begin with a "#" symbol. White space may follow, and then a keyword must be distinguishable. Depending on the keyword, there may be parameters; the rest of the command line can be quite arbitrary. The following commands are supported:

You may nest *include* to any depth, assuming the runtime support for this program is reasonable. Once the end of an *included* file is reached, processing continues after the *include* command causing the file inclusion. If a file cannot be opened or found, a message is printed, but processing continues.

The *line* command is intended for program generators. For purposes of diagnostics and position stamps in the output file, *p* will accept a line number and optionally a filename from the *line* command.

Other lines starting in "#" are processed as if they were text. *p* will thus pass through such things as "asm" and "endasm," provided these lines are not modified by macro expansions. Position stamps from a previous run of *p*

would also be passed through again.

Implementation

The task of preprocessing can be nicely structured into four problems, each solved essentially by a single C function:

```
take care of start options, initialize

while (there is another line)

    if (after removing comments, there
    is something && 

        after observing commands, there
        is something)

            preprocess the text line
```

main() takes care of start options and initialization. getline() collects a nonblank line and takes care of line continuation. comment() eliminates comments, which may extend over several lines, and removes leading white space. command() knows whether you are currently skipping in observance of some *if* construct; if you are, command() pretends that a command was actually found so that the input line is not processed further. If there is a preprocessor command, command() will recognize and execute it. If the current line is regular text, process() takes care of macro expansion and output.

getline() simply collects input characters into a buffer until a newline character or end of file is found. If there is a backslash followed by a newline, both characters are ignored (but source lines are counted). If there is end of file following a backslash, getline() complains. At end of file, getline() attempts to pop the stack of open input files. getline() returns once it encounters a nonblank input line or if the end of the initial input file is reached. Non-ASCII characters are not accepted as input to simplify subsequent processing.

For input and output two buffers must be maintained and should be able to grow more or less without limit. The rebuff() routine, which is called with a full buffer, handles this. It will relocate the buffer and copy the information over. Clearly, no other pointers into the buffer should exist at that point.

The comment() algorithm is quite simple: it copies everything until /* then quits copying (and reporting that there is text material) until */ is found. Matters are complicated slightly by strings, (invalid) character constants, and backslashes. The problem is handled with the state variable "cmode," which maintains the current context — comment, string, character constant — across calls. comment() will suppress blank lines, as well as initial white space.

command() deals with preprocessor commands in all those line buffers that comment() did not prevent from being passed on. Calls to the symbol table management routines handle command processing. *if* constructs are implemented through two variables: iflevel, which counts the current nesting depth of these constructs, and skip. skip is usually and initially zero, indicating that text should be processed. If text should be ignored (due to some *if* or *else*), skip is set to the value of iflevel at which skipping should terminate. If you are skipping and reach *else* or *endif* at the proper level, you are done; skip reverts to zero.

There really should be only one *else* per *if*. Enforcing this would require a stack to indicate at each "iflevel" if we have already seen *else* or not. I felt that this was not really necessary — consequently (as in Hendrix's compiler), multiple *else* are allowed.

line and *include* require a certain amount of data processing. The syntax must be verified, and the relevant information must be stacked. There are stacks of open file pointers, filenames, and last line numbers. All the stacks are handled by push routines, which return the address of a new element linked before the stack. The result of push, therefore, must always be assigned back to the stack top pointer, which is passed as an argument. A common pop() routine handles removal of unwanted elements. A string stack is also used to hold the *include* prefixes passed as options.

process() does the actual work if a line is ever passed on to it. The line buffer is scanned and copied to the oline buffer. The outmacro(), outnumber(), outdelim(), and out() routines are called to manage processing of a macro call, a numerical constant, a string or character constant, and a simple character, respectively. As long as a macro was actually expanded in a pass over line, the two buffers are interchanged and processing is repeated until either no more expansion is performed or a count runs out (to prevent infinite recursion). output() takes care of emitting position stamps and the processed text.

Finding names in the symbol table and undefining them is quite simple: the symbol table is a one-way linked list of entries, each containing a pointer to the string defining the symbol and the symbol name itself. If the symbol is a parameterized macro, the name is followed by a left parenthesis and a (binary) parameter count. find() must take care to match the names correctly. If a symbol definition is parameterized, the value contains (binary) parameter numbers in place of the original parameter names. Each parameter number takes just one byte and is flagged in the high bit since all other text is ASCII.

The symbol table can employ a hashing mechanism. If the symbolic name HASH is undefined, the symbol table consists of a single linear list. If the name is defined, it must be as a power of 2, designating how many such linear lists should be kept. In this case each name is first converted to a number indicating which of the lists should be used. This simple mechanism and a rather naive hash function in find() cut preprocessing time by about 20 percent.

define() does a significant amount of text processing to prepare a parameterized macro definition. marknm() separates the macro header from the macro value within the line buffer. ismacro() and isname() make sure that the macro header uses only appropriate symbols and the proper arrangements of commas and parentheses. ismacro() prepares yet another string stack of parameter names. define() then removes leading and trailing white space from the value and replaces parameter names by flagged numbers. Finally you must guard against redefinitions and store the result. A redefinition is flagged only if it is truly different — therefore, header files usually can be read several times without complaint.

outmacro() also does a large amount

of text processing. If the macro is parameterized, markarg() is used to flag the text arguments in the input buffer and to prepare a stack of pointers to these argument values. This task, too, is complicated by the usual potential assortment of parentheses and string delimiters. Once the argument list is collected, it is a simple matter (possibly handled with outarg()) to emit the macro definition with the argument texts in place.

Installation

Getting *p* to run on your system might be a bit tricky. There is a bootstrap problem — *p* uses features supported only by *p* and not by Hendrix's compiler — and there is a problem concerning the runtime support.

Overcoming the bootstrap problem is actually quite simple. You should replace *double quotes* as a character constant by the value 34 (my Small C compiler got confused in certain places until I did this). You need to replace all constants that your compiler does not support (e.g., the definitions of PARM and PARMNO and the values for base in the routine outnumber()). You might have to replace the character constants in the routine outdelim() if your compiler does not yet support those escape sequences. Finally, you will have to play preprocessor for those macros that are parameterized. (Yes, it was not nice to use those, but I did want to show how parameterized macros can be used to clarify data types.)

The runtime support is quite a different matter. While I agree with Jim Hendrix that these matters ought to be standardized, I am much in favor of programming on my CP/M system at home just as I do on our Unix systems at the office. I have actually made a runtime support that looks like the standard libraries available with Unix version 7 and above, is based on CP/M, and supports all BIOS and BDOS calls from Small C.

I have had access to chapter 17 of Hendrix's book on Small C, describing his runtime support. While I did not have access to the runtime support itself and therefore could not test it, I believe that installing *p* should be quite simple. The following probably must be done:

FILE	should be defined as *int*.
__drive()	needs to access BDOS function 25.
__narg()	is supported by the compiler.
index()	is Hendrix's strchr().
itod()	can be coded using Hendrix itod().

I process the arguments to main() directly. Depending on the actual implementation, you might have to use Hendrix's function getar().

I am assuming that the storage allocator, calloc()/cfree(), supports random order release of memory. The code in Listing Two (page 81) may be useful to those wishing to add such random order release to the Hendrix-Payne library published in the May and June 1984 issues of *DDJ*.

Meanwhile, you probably should consult Kernighan and Ritchie's book to learn about all the routines that are mentioned "extern" at the beginning of the program. Most of them are quite simple to construct. It is essential, however, that you provide multiple open files so that arbitrary nesting of file inclusion is possible. You will also need a memory allocator, calloc() and cfree(), that will reuse available space and is reasonably stable. I am using a scheme where memory above the load module and below the stack is managed by a list of words, each word pointing to the next. The low bit in each such word indicates if the area past that word and up to the next one is allocated or free. Kernighan and Ritchie mention, and Unix supports, routines to classify characters. This is most easily implemented as a 128-byte table with each byte classifying the corresponding character as special, upper or lower case, numeric, hexadecimal, white space, punctuation, or control character. The routines are then simple masking operations that can now be provided as macros. ■■J

Part of this work was done during a sabbatical spent at the University of Illinois; in particular, the Small C system was obtained from "UseNet."
p *can be compiled with the C compiler on Unix/vii.*

(Listings begin on page 52)

P (Text begins on page 46)

Listing One

```
/*
 *      p -- small C preprocessor
 *      ats 6/83
 */

/*
 *      define...
 *
 *      verbose         to support -v for debugging
 */

char usage[] =
#ifdef  verbose
        "p [-d n[=v]] [-e] [-i d:] [-u n] [-v] [in [out]]",
        vflag;          /* set by -v */
#else
        "p [-d n[=v]] [-e] [-i d:] [-u n] [in [out]]";
#endif

                        /* command names:                */
#define DEFINE  1       /* # define name text            */
                        /* # define name(name,...) text */
#define ELSE    2       /* # else                        */
#define ENDIF   3       /* # endif                       */
#define IFDEF   4       /* # ifdef name                  */
#define IFNDEF  5       /* # ifndef name                 */
#define INCLUDE 6       /* # include "file"              */
                        /* # include <file>              */
#define LINE    7       /* # line number name           */
#define UNDEF   8       /* # undef name                  */
#define DEFAULT 0       /* any others passed on          */

/*
 *      "FEATURES":
 *
 *              Macro calls must be fully contained on one source
 *              line -- all lines can be continued with \, however.
 *
 *              Recursive definitions are not detected as such.
 *              'p' will report as per NEST.
 *
 *              #else can be used (to reverse the current
 *              #if condition) arbitrarily often.
 */

#include <stdio.h>

/*
 *      i/o header file

#define FILE    ???     type to represent files (used as FILE*)
#define stdin   ???     pre-opened standard input file
#define stdout  ???     pre-opened standard output file
#define stderr  ???     pre-opened diagnostic output file
#define NULL    0       null pointer, false
#define EOF     ???     end of file indication
 */

#include <ctype.h>
```

```
/*
 *      character classification macros header file
 *
 *      isascii(i)      i is ASCII character
 *      isalnum(c)      c is letter or digit
 *      isalpha(c)      c is letter
 *      isdigit(c)      c is digit
 *      islower(c)      c is lower case letter
 *      isspace(c)      c is white space
 *      isupper(c)      c is upper case letter
 *      isxdigit(c)     c is base 16 digit
 *
 *      i can be any integer, isascii(c) must be true for c
 */

#define INCR    80      /* line buffer increment */
#define HASH    128     /* hash table size (power of 2) */
#define NEST    10      /* limit for reprocessing - -1 is "infinite" */

                        /* cmode states */
#define CMcmt   1       /* in comment */
#define CMstr   2       /* in string */
#define CMchr   3       /* in character constant */

#define PARM    0x80    /* flag macro parameter number */
#define PARMNO  0x7f    /* extract parameter number */

/*
 *      special data types
 */

#define LIST        int             /* list of word or string values */
#define l_next(x)   (*(x))          /* -> next element */
#define l_word(x)   ((x)[1])        /* word value */
#define l_str(x)    ((x)+1)         /* -> string value */
#define sz_WORD     4               /* size of word list element */
#define sz_STR(s)   (3+strlen(s))   /* size of string list element */

#define SYMBOL      int             /* list of symbol table elements */
#define s_next(x)   (*(x))          /* -> next element */
#define s_val(x)    ((x)[1])        /* -> defined value */
#define s_name(x)   ((x)+2)         /* -> name */
#define sz_SYM(n)   (5+strlen(n))   /* size of symbol table element */

/*
 *       runtime support routines
 */

extern  _drive(),       /* BDOS function 25: current drive number */
        _narg(),        /* number of arguments passed in this call */
        calloc(),       /* (n,l) return NULL or -> n elements of length l */
        cfree(),        /* (p) free area at p, returned by calloc() */
        exit(),         /* terminate program execution */
        fclose(),       /* (f) close file described by f */
        fgetc(),        /* (f) return EOF or next character from file f */
        fopen(),        /* (n,m) return NULL or descriptor for file "n"
                           opened to read (m == "r"), write ("w"),
                           or append ("a") */
        fputc(),        /* (c,f) write c on file f, return EOF or c */
        fputs(),        /* (s,f) write string s on file f */
        freopen(),      /* (n,m,f) like fopen(), but close and reuse f */
        index(),        /* (s,c) find c in string s, return NULL or -> to it.
                           '\0' is always found */
        itod(),         /* (i) return -> (static) string with i in decimal */
```

(Continued on next page)

P (Listing Continued, text begins on page 46)

Listing One

```
        strcmp();       /* (a,b) <, ==, > 0 as string a is <, ==, > string b */
        strcpy();       /* (a,b) copy string b to string a */
        strlen();       /* (s) return number of characters in string s */
        strncmp();      /* (a,b,n) like strcmp(), but for n bytes at most */
        strncpy();      /* (a,b,n) like strcpy(), but for n bytes at most */

/*
 *      global data
 */

int     parmno,         /* current number of parameters */
        linelen,        /* current maximum usable length */
        olinelen,
        lineno,         /* current line number */
        olineno,
        iflevel,        /* depth of open #if */
        skip;           /* non-0: iflevel to skip to */

char    eflag,          /* set by -e: prevent position stamps */
        cmode,          /* comment() mode */
        *line,          /* dynamic input line buffer */
        *lp,            /* current position in line */
        *oline,         /* dynamic output line buffer */
        *olp;           /* current position in oline */

LIST    *drive,         /* include prefixes */
        *filenms,       /* open file names */
        *files,         /* open file pointers */
        *lines,         /* line numbers */
        *parms;         /* parameters */

SYMBOL  *symbol;        /* list of symbol table elements */
#ifdef  HASH            /* hash feature (optional) */
                        /* symbol set by find() to -> hashtab at s */
int     hashtab[HASH];  /* really SYMBOL *: begin of chains */
#endif

FILE    *infile;        /* current input file */

main(argc, argv)
        int argc;
        int *argv;
{       char *cp, *vp;

#ifdef  verbose
        LIST *ip;
#endif

        /* current drive is last include prefix */
        vp = "a:";
        *vp += _drive();
        drive = pushs(drive, vp);

        /* predefine "cpm" */
        define("cpm", "");

        /* process arguments, values may be joined or separate */
        while (--argc)
        {       cp = *++argv;
                if (*cp != '-')
```

```
                        break;
                switch (cp[1]) {
                case 'd':
                        if (*(cp += 2))
                                ;
                        else if (--argc == 0)
                                goto error;
                        else
                                cp = *++argv;
                        if (vp = index(cp, '='))
                                *vp++ = '\0';
                        else
                                vp = "";
                        define(cp, vp);
                        break;
                case 'e':
                        eflag = 1;
                        break;
                case 'i':
                        /* explicit prefixes in order right to left */
                        if (*(cp += 2))
                                ;
                        else if (--argc == 0)
                                goto error;
                        else
                                cp = *++argv;
                        drive = pushs(drive, cp);
                        break;
                case 'u':
                        if (*(cp += 2))
                                ;
                        else if (--argc == 0)
                                goto error;
                        else
                                cp = *++argv;
                        undefine(cp);
                        break;
#ifdef  verbose
                case 'v':
                        vflag = 1;

                        break;
#endif
                default:
                        goto error;
                }
        }

        /* input file drive is first include prefix */
        vp = "a:";
        *vp += _drive();

        /* allow input and output files */
        switch (argc) {
        case 2:         /* use input, drive(input), output */
        case 1:         /* use input, drive(input) */
                if (cp[1] == ':')
                {       vp = "?:";
                        *vp = cp[0];
                }
```

(Continued on next page)

P (Listing Continued, text begins on page 46)

Listing One

```
                if (freopen(*argv, "r", stdin) == NULL)
                {       where("cannot read", *argv);
                        exit();
                }
                filenms = pushs(filenms, *argv);
#ifdef  verbose
                if (vflag)
                        where("reading");
#endif
                if (--argc)
                {       if (freopen(*++argv, "w", stdout) == NULL)
                        {       filenms = NULL;
                                where("cannot write", *argv);
                                exit();
                        }
#ifdef  verbose
                        if (vflag)
                                where("writing", *argv);
#endif
                }
        case 0:         /* use stdin, current drive */
                break;
        default:
error:          where(usage);
                exit();
        }

        /* set first include prefix */
        drive = pushs(drive, vp);

#ifdef  verbose
        if (vflag)
        {       for (ip = drive; ip; ip = l_next(ip))
                        where("drive", l_str(ip));
        }
#endif

        /* start reading on stdin */
        infile = stdin;

        /* allocate first buffers */
        if ((line = calloc(INCR, 1)) == NULL
            || (oline = calloc(INCR, 1)) == NULL)
        {       where("no room");
                exit();
        }
        olinelen = linelen = INCR;

        /* make sure, we first get a position stamp */
        olineno = lineno - 3;

        /* main loop */
        while (getline())
                if (! comment() && ! command())
                        process();
}

getline()               /* line = complete line, ascii */
                        /* return false on EOF */
{       int c;          /* current character */
```

```
        /* move to lp, concatenating continued lines */
        for (lp = line; ; )
        {       switch (c = fgetc(infile)) {
                case '\\':
                        switch (c = fgetc(infile)) {
                        case EOF:
                                where("trailing \\");
                        case '\n':
                                ++lineno;
                                continue;
                        }
                        in('\\');
                default:
                        if (! isascii(c))
                                where("illegal character");
                        else
                                in(c);
                        continue;
                case EOF:
                        ++ lineno;
                        if (lp != line)
                                break;
                        else if (files)
                        {       fclose(infile);
#ifdef  verbose
                                if (vflag)
                                        where("end include");
#endif
                                infile = pop(&files);
                                lineno = pop(&lines);
                                olineno = lineno - 3; /* stamp! */
                                pop(&filenms);

                                continue;
                        }
                        return 0;
                case '\n':
                        ++lineno;
                        if (lp == line)
                                continue;
                }
                break;  /* got a nonempty line */
        }
        *lp = '\0';
#ifdef  verbose
        if (vflag)
                where("getline", line);
#endif
        return 1;
}

comment()               /* line = line w/out comments, lead white space */
                        /* return true if comment line */
{       char c;

        /* move from olp to lp, eliminating comments */
        for (lp = olp = line; ; )
        {       switch (c = *olp++) {
                case '\\':
                        if (cmode != CMstr && cmode != CMchr)
                                break;
                        in(c);
                        if (c = *olp++)
                                break;
```

(Continued on page 60,

P (Listing Continued, text begins on page 46)

Listing One

```
                case '\0':
                        if (cmode == CMstr)
                        {       if (! skip)
                                        where("unbalanced \"");
                                in('\"');
                                cmode = 0;
                        }
                        else if (cmode == CMchr)
                        {       if (! skip)
                                        where("unbalanced \'");
                                in('\'');
                                cmode = 0;
                        }
                        *lp = '\0';
#ifdef  verbose
                        if (vflag)
                                where("comment", line);
#endif
                        return lp == line;
                case '/':
                        if (cmode == 0 && *olp == '*')
                        {       cmode = CMcmt;
                                ++olp;
                                if (lp != line)
                                        in(' ');
                                continue;
                        }
                        break;
                case '*':
                        if (cmode == CMcmt && *olp == '/')
                        {       cmode = 0;
                                ++olp;
                                continue;
                        }
                        break;
                case '\"':
                        switch (cmode) {
                        case 0:
                                cmode = CMstr;
                                break;
                        case CMstr:
                                cmode = 0;
                        }
                        break;
                case '\'':
                        switch (cmode) {
                        case 0:
                                cmode = CMchr;
                                break;
                        case CMchr:
                                cmode = 0;
                        }
                }
                if (cmode != CMcmt && (! isspace(c) || lp != line))
                        in(c);
        }
}

command()               /* process commands */
```

```
                        /* return true if done (i.e., to skip) */
{       int k;
        LIST *dp;
        FILE *fp;
        char *cp;

/*
 *      #if algorithm
 *
 *      skip    if non-zero, knows #if-level to which to skip;
 *              while skipping, comment() is executed, but not process().
 *
 *      iflevel current nesting depth of #if;
 *              counted even while skipping (of course).
 *
 *      #else   if skipping to current #if-level, stop skipping;
 *              if not skipping, start skipping to current level.
 *
 *      In order to limit #else to at most one per #if, we
 *      would need a stack; why bother??
 */
                if (*line != '#' || (k = kind(&lp)) == DEFAULT)
                        return skip;

                /* process the command */
                switch (k) {
                case DEFINE:
                        if (! skip)
                                define(lp, marknm(lp));
                        break;
                case ELSE:
                        if (! skip && iflevel == 0)
                                where("#else without #if");
                        else if (skip == iflevel)
                                skip = 0;
                        else if (skip == 0)
                                skip = iflevel;
                        break;
                case ENDIF:
                        if (! skip && iflevel == 0)
                                where("#endif without #if");
                        else
                        {       if (skip == iflevel)
                                        skip = 0;
                                --iflevel;
                        }
                        break;
                case IFDEF:
                case IFNDEF:
                        ++iflevel;
                        if (! skip)
                        {       marknm(lp);
                                if (isname(lp,""))
                                        if (find(lp))
                                        {       if (k == IFNDEF)
                                                        skip = iflevel;
                                        }
                                        else
                                        {       if (k == IFDEF)
                                                        skip = iflevel;
                                        }
                        }
                        break;
                case INCLUDE:
                        if (! skip)
```

(Continued on next page)

P (Listing Continued, text begins on page 46)

Listing One

```
                        if (! markfl(lp))
                                where("#include?");
                        else if (lp[2] == ':')
                        {       ++lp;
                                if (fp = fopen(lp, "r"))
                                {
pushfile:                               files = pushw(files, infile);
                                        lines = pushw(lines, lineno);
                                        lineno = 0;
                                        olineno = lineno - 3; /* stamp ! */
                                        filenms = pushs(filenms, lp);
                                        infile = fp;
#ifdef  verbose
                                        if (vflag)
                                                where("including", lp);
#endif
                                }
                                else
                                        where("cannot open include file", lp);
                        }
                        else
                        {       dp = drive;
                                if (*lp == '(')
                                        dp = l_next(dp);
                                *lp-- = ':';
                                for (; dp; dp = l_next(dp))
                                {       *lp = *l_str(dp);
                                        if (fp = fopen(lp, "r"))
                                                goto pushfile;
                                }
                                where("cannot find include file", lp+2);
                        }
                break;
        case LINE:
                if (! skip)
                {       if (isdigit(*lp))
                        {       for (k = *lp - '0'; isdigit(*++lp); )
                                        k = k*10 + *lp-'0';
                                while (isspace(*lp))
                                        ++lp;
                                cp = lp;
                                while (*lp && ! isspace(*lp))
                                        ++lp;
                                if (lp != cp)
                                {       *lp = '\0';
                                        lineno = k;
                                        if (filenms)
                                                pop(&filenms);
                                        filenms = pushs(filenms, cp);
                                        break;
                                }
                        }
                        where("#line?");
                }
                break;
        case UNDEF:
                if (! skip)
                {       marknm(lp);
                        undefine(lp);
```

```
                }
        }
        return 1;
}

process()               /* process regular input line */
{       char expand;    /* reprocess flag */
        char c;         /* current input character */
        SYMBOL *sp;     /* -> found symbol */
        char *name;     /* -> begin of name */
        int i;
        int nest;

        /* expand one buffer into the other */
        nest = NEST+1;
        do
        {       lp = line;
                *(olp = oline) = '\0';
                expand = 0;
                while(c = *lp++)
                        if (isalpha(c) || c == '_')
                        {       name = out(c);
                                while ((c = *lp) && (isalnum(c) || c == '_'))
                                {       out(c);
                                        ++lp;
                                }
                                if (sp = find(name))
                                {       expand = 1;
                                        outmacro(name, sp);
                                }
                        }
                        else if (isdigit(c))
                                lp = outnumber(--lp);
                        else if (c == '\'' || c == 34)
                                lp = outdelim(--lp);
                        else
                                out(c);
                /* if something changed, flip buffers */
                if (expand)
                {       lp = line;
                        line = oline;
                        oline = lp;
                        i = linelen;
                        linelen = olinelen;
                        olinelen = i;
                }
        } while (expand && --nest);
        if (expand)
                where("#define nested too deep");
        output(oline);
}

/*
 *      symbol table routines
 */

define(s, v)            /* #define s v */
        char *s;        /* name of symbol ?? */
        char *v;        /* value */
{       SYMBOL *r;
        int f;
        char *cp, *p, c, *name;

        if (! ismacro(s))
                return;
```

(Continued on page 66)

P (Listing Continued, text begins on page 46)

Listing One

```
        /* prune and parametrize value */
        while (isspace(*v))
                ++v;
        if (*v)
        {       for (cp = v + strlen(v); cp != v; )
                        if (! isspace(*--cp))
                                break;
                        else
                                *cp = '\0';
                /* if we have parameters, replace names by positions */
                if (parmno)
                {       p = cp = v;
                        while (c = *cp++)
                                if (isalpha(c) || c == '_')
                                {       *(name = p++) = c;
                                        while ((c = *cp)
                                                && (isalnum(c) || c == '_'))
                                        {       *p++ = c;
                                                ++cp;
                                        }
                                        if (f = findparm(name, p-name))
                                        {       *(p = name) = f | PARM;
                                                ++p;
                                        }
                                }
                                else
                                {       *p++ = c;
                                        /* name as trailer of a constant?? */
                                        if (c == '0'
                                                 && (*cp == 'x' || *cp == 'X'))
                                                do
                                                        *p++ = *cp++;
                                                while (isxdigit(*cp));
                                        else if (isdigit(c))
                                                while (isdigit(*cp))
                                                        *p++ = *cp++;
                                        else if (c == '\\')
                                                if (*cp)
                                                        *p++ = *cp++;
                                }
                        *p = '\0';
                }
        }

        /* check if (different) redefinition */
        if (r = find(s))
        {       if (strcmp(s_val(r), v) != 0)
                        where("redefining", s);
                undefine(s);    /* parmno may change */
#ifdef  verbose
                if (vflag)
                        fputs("redefine ", stderr);
#endif
        }
        /* if parametrized, save count */
        if (parmno)
        {       cp = s + strlen(s);
                *cp = '(';
                *++cp = parmno;
```

```
                *++cp = '\0';
        }


        /* ready to make new entry */
        if ((r = calloc(sz_SYM(s), 1)) == NULL)
        {       where("no room");
                exit();
        }
        else
        {
#ifdef  HASH    /* find() sets symbol -> hashtab at s */
                s_next(r) = *symbol;
                *symbol = r;
#else
                s_next(r) = symbol;
                symbol = r;
#endif
                s_val(r) = NULL;
                strcpy(s_name(r), s);
#ifdef  verbose
                if (vflag)
                        fputs("define ", stderr);
#endif
        }


        /* save new value */
        if ((s_val(r) = calloc(strlen(v)+1, 1)) == NULL)
        {       where("no room");
                exit();
        }
        strcpy(s_val(r), v);
#ifdef  verbose
        if (vflag)
        {       fputs(s, stderr);
                fputc(' ', stderr);
                fputs(s_val(r), stderr);
                fputc('\n', stderr);
        }
#endif
}

undefine(s)             /* #undef s */
        char *s;        /* name of symbol ?? */
{       SYMBOL *r, *p;

        if (isname(s, "") && (r = find(s)))
        {       cfree(s_val(r));
                /* need to unlink symbol descriptor from chain */
#ifdef  HASH    /* find() sets symbol -> hashtab at s */
                if (r == *symbol)
                        *symbol = s_next(*symbol);
                else
                {       for (p = *symbol;
#else
                if (r == symbol)
                        symbol = s_next(symbol);
                else
```

(Continued on next page)

P (Listing Continued, text begins on page 46)

Listing One

```
                {       for (p = symbol;
#endif
                                        s_next(p) != r; p = s_next(p))
                                ;
                        s_next(p) = s_next(r);
                }
                cfree(r);
#ifdef  verbose
                if (vflag)
                        where("undefine", s);
#endif
        }
}

find(s)                 /* locate s in symbol table */
                        /* return NULL or -) entry */
        char *s;        /* name to find */
{       SYMBOL *r;
        char *sp, *rp, c;
#ifdef  HASH
        int h;

        /* symbol table chains start in hashtab[] */
        /* compute hash address as sum of letters */
        for (h = 0, sp = s; c = *sp; ++sp)
                h += c;
        symbol = hashtab + (h & (HASH-1));

        /* run down the chain */
        for (r = *symbol;
#else
        /* symbol table chain is one linear list */
        /* run down the chain */
        for (r = symbol;
#endif
                        r; r = s_next(r))
        {       for (sp = s, rp = s_name(r); (c = *sp) && *rp == c; ++sp, ++rp)
                        ;
                if (c == '\0' && (*rp == '\0' || *rp == '('))
                        return r;
        }
        return NULL;
}

findparm(s, l)          /* return 0 or parameter number */
        char *s;        /* -) begin of possible parameter name */
        int l;          /* length of name */
{       int f;
        LIST *p;

        for (f = 0, p = parms; p; ++f, p = l_next(p))
                if (strncmp(l_str(p), s, l) == 0)
                        return parmno - f;
        return 0;
}

isname(s,d)             /* true, if s is a name */
                        /* return -) delimeter or NULL */
        char *s;        /* -) begin of name */
        char *d;        /* chars in which name may also end */
```

```
{       char *cp, c;

        for (cp = s; index(d, c = *cp) == NULL; ++cp)
                if (! isalnum(c) && c != '_')
                        goto error;
        if (cp == s || isdigit(*s))
        {
error:          where("illegal name", s);
                return NULL;
        }
        return cp;      /* return -> delimeter */
}

ismacro(s)              /* true, if s is a macro header */
        char *s;        /* -> begin of name or header */
{       char *cp, c;

        while (parms)   /* free old parameter list */
                pop(&parms);
        parmno = 0;
        if ((s = isname(s, "(")) == NULL)
                return 0;
        if (*s)         /* we have a new macro */
        {       *s = '\0';      /* delimit name */
                do              /* and parse parameters */
                {       while (isspace(*++s))
                                ;
                        if (cp = isname(s, ",) \t"))
                        {       c = *cp;
                                *cp = '\0';
                                parms = pushs(parms, s);
                                ++ parmno;
                        }
                        else
                                return 0;
                        while (isspace(c))
                                c = *++cp;
                        s = cp;
                } while (c == ',');
                if (c != ')')
                {       where("illegal macro header");
                        return 0;
                }
        }
        return 1;
}

marknm(s)               /* bypass and terminate macro header */
                        /* return -> value */
        char *s;        /* ->begin of name */
{       char c;

        /* find white space or ( */
        while ((c = *s) && ! isspace(c) && c != '(')
                ++s;

        /* if (, there must be names, white space and then ) */
        if (c == '(')
        {       while ((c = *++s) && c != ')')
                        ;
                /* after ) there must be \0 or white space */
                if (c && (s[1] == '\0' || isspace(s[1])))
                        ++s;
        }
```

(Continued on page 72)

P (Listing Continued, text begins on page 46)

Listing One

```
        /* terminate in place of white space */
        if (*s)
                *s++ = '\0';

        /* this is a rough draft -- see ismacro/isname */
        return s;
}

/*
 *      input and output routines
 */

in(c)                   /* store incoming character */
        char c;         /* to be stored at lp */
{
        *lp++ = c;
        if (lp >= line+linelen)
                rebuff(&lp, &line, &linelen);
}

out(c)                  /* store a character, return -> stored char */
        char c;         /* to be stored at olp */
{
        *olp++ = c;
        if (olp >= oline+olinelen)
                rebuff(&olp, &oline, &olinelen);
        *olp = '\0';    /* maintain trailer */
        return olp-1;
}

rebuff(p, buf, len)     /* make buffer longer */
        int *p;         /* & current pointer */
        int *buf;       /* & buffer pointer */
        int *len;       /* & maximum length */
{
        if ((*p = calloc(*len + INCR, 1)) == NULL)
        {       where("no room");
                exit();
        }
        strncpy(*p, *buf, *len);
        cfree(*buf);
        *buf = *p;
        *p = *buf + *len;
        *len += INCR;
}

output(s)               /* write a string */
        char *s;        /* to write as a line */
{
        /* synchronize output linecount */
        if (! eflag && ++olineno != lineno)
        {       if (++olineno != lineno)
                {       fputc('#', stdout);
                        fputs(itod(olineno = lineno), stdout);
                        if (filenms)
                        {       fputc(' ', stdout);
                                fputs(l_str(filenms), stdout);
                        }
                }
```

```
                fputc('\n', stdout);
        }

        /* emit string as a line */
        fputs(s, stdout);
        if (fputc('\n', stdout) == EOF)
        {       where("output file full");
                exit();
        }
}

/*
 *      C constant processing:
 *
 *      digits          decimal
 *      0digits         octal
 *      0xdigits        hexadecimal
 *      'c'             character value (escapes ok)
 */

outnumber(cp)           /* store a C constant in decimal */
                        /* return -> past it */
        char *cp;       /* -> constant text (digit) */
{       char c, *p;
        int base;
        int i;

        base = 10;
        i = 0;
        if ((c = *cp) == '0')
        {       base = 010;
                if ((c = *++cp) == 'x' || c == 'X')
                {       base = 0x10;
                        c = *++cp;
                }
        }
        for ( ; c; c = *++cp)
        {       if (isdigit(c))
                        c -= '0';
                else if (isxdigit(c))
                        if (isupper(c))
                                c -= 'A' - 10;
                        else
                                c -= 'a' - 10;
                else
                        break;
                if (c < base)
                        i = i*base + c;
                else
                        break;
        }
        for (p = itod(i); c = *p; ++p)
                out(c);
        return cp;
}

outdelim(cp)            /* store a delimited string, return -> past trailer */
        char *cp;       /* -> delimeter */
{       char c, *p;

        if ((c = *cp) == '\"')
        {       out(c);
                while (c = *++cp)
                {       out(c);
                        if (c == '\"')
```

(Continued on next page)

P *(Listing Continued, text begins on page 46)*

Listing One

```
                        return cp+1;
                if (c == '\\')
                        if (c = *++cp)
                                out(c);
                        else
                                break;
        }
    }
    else    /* it must be character constant */
        switch (c = *++cp) {
        case 0:
        case '\'':
                goto error;
        case '\\':
                switch (c = *++cp) {
                case 'b':  c = '\b'; break;
                case 'f':  c = '\f'; break;
                case 'n':  c = '\n'; break;
                case 'r':  c = '\r'; break;
                case 't':  c = '\t';
                case '\'':
                case '\\':
                case '\"':              break;
                default:
                        if (! isdigit(c) || (c -= '0') > 7)
                                goto error;
                        if (isdigit(cp[1]) && cp[1] <= '7')
                        {       c = (c << 3) + *++cp - '0';
                                if (isdigit(cp[1]) && cp[1] <= '7')
                                        c = (c << 3) + *++cp - '0';
                        }
                }
        default:
                if (*++cp != '\'')
                {
error:                  where("illegal character constant");
                        while (*cp && *cp != '\'')
                                ++cp;
                        if (*cp)
                                ++cp;
                        break;
                }
                for (p = itod(c); *p; ++p)
                        out(*p);
                out(' ');
                ++cp;
        }
    return cp;
}

/*
 *      macro processing
 */

outmacro(at, s)         /* replace string by macro value */
    char *at;           /* replace from here on */
    SYMBOL *s;          /* using this definition */
{   char *vp, c;
```

```
        /* set output up for replacement */
        olp = at;

        /* force white space around replacement */
        if (olp ) oline && (! isspace(olp[-1])))
                out(' ');

        /* if parametrized, collect arguments */
        if (vp = index(s_name(s), '('))
                markarg(*++vp);
        else
                parmno = 0;

        /* emit replacement */
        for (vp = s_val(s); c = *vp++; )
                if (c & PARM)
                        outarg(c & PARMNO);
                else
                        out(c);
        /* white space */
        out(' ');
}

markarg(n)              /* mark and collect arguments */
        int n;          /* number to find */
{       char c, cmode;  /* cmode during argument collection only */
        int lpar;

        /* release parameter/argument list, if any */
        while (parms)
                pop(&parms);
        parmno = 0;

        /* find ( */
        while (isspace(c = *lp))
                ++lp;

        /* collect */
        if (c == '(')
        {       do
                {       parms = pushw(parms, ++lp);
                        ++ parmno;
                        lpar = cmode = 0;
                        for ( ; c = *lp; ++lp)
                        {       switch (c) {
                                case '(':
                                        if (cmode == 0)
                                                ++ lpar;
                                        continue;
                                case ',':
                                        if (cmode || lpar)
                                                continue;
                                        break;
                                case ')':
                                        if (cmode || lpar --)
                                                continue;
                                        break;
                                case '\'':
                                        switch (cmode) {
```

(Continued on next page)

P (Listing Continued, text begins on page 46)

Listing One

```
                                        case 0:
                                                cmode = CMchr;
                                        case CMstr:
                                                continue;
                                        }
                                        cmode = 0;
                                        continue;
                                case '\"':
                                        switch (cmode) {
                                        case 0:
                                                cmode = CMstr;
                                        case CMchr:
                                                continue;
                                        }
                                        cmode = 0;
                                        continue;
                                case '\\':
                                        if (*++lp == '\0')
                                                break;
                                default:
                                        continue;
                                }
                                *lp = '\0';
                                break;
                        }
                } while (c == ',');
                if (c == ')')
                        ++lp;
                else
                        where("incomplete macro call");
        }

        /* check and fill argument count */
        if (parmno != n)
                where("wrong number of arguments");
        for ( ; parmno < n; ++parmno)
                parms = pushw(parms, "");
}

outarg(i)               /* emit argument */
        int i;          /* number to emit */
{       LIST *p;
        char *cp, c;

        /* play double safe */
        if (i > parmno)
        {       where("outarg()))??");
                exit();
        }

        /* locate */
        for (i = parmno-i, p = parms; i && p; --i, p = l_next(p))
                ;
        if (p == NULL)
        {       where("outarg(NULL)??");
                exit();
        }

        /* emit, no white space */
```

```
        for (cp = l_word(p); c = *cp; ++cp)
                out(c);
}

/*
 *      stack routines
 */

pushw(l, w)                     /* push word, return -> new list */
        LIST *l;                /* list */
        int w;                  /* word to push */
{       LIST *r;
        if ((r = calloc(sz_WORD, 1)) == NULL)
        {       where("no room");
                exit();
        }
        l_next(r) = l;
        l_word(r) = w;
        return r;
}

pushs(l, s)                     /* push string, return -> new list */
        LIST *l;                /* list */
        char *s;                /* string to push */
{       LIST *r;

        if ((r = calloc(sz_STR(s), 1)) == NULL)
        {       where("no room");
                exit();
        }
        l_next(r) = l;
        strcpy(l_str(r), s);
        return r;
}

pop(l)                          /* pop list, return word */
        LIST *l;                /* really **: list header */
{       LIST *r;
        int i;

        if (*l == NULL)
        {       where("pop(NULL)??");
                exit();
        }
        r = *l;                 /* element to pop */
        i = l_word(r);          /* result */
        r = l_next(r);          /* following element */
        cfree(*l);
        *l = r;
        return i;               /* nonsense for a string list */
}

/*
 *      other utilities
 */

where(vararg)                   /* error message writer */
        int vararg;             /* arbitrarily many strings */
{       int narg, *argv;

        narg = _narg();
        argv = &vararg;
        argv += narg;
        if (filenms)
```

(Continued on page 80)

P (Listing Continued, text begins on page 46)

Listing One

```
        {       fputs(l_str(filenms), stderr);
                if (lineno)
                        fputs(", ", stderr);
                else
                        fputs(": ", stderr);
        }
        if (lineno)
        {       fputs("line ", stderr);
                fputs(itod(lineno), stderr);
                fputs(": ", stderr);
        }
        while (narg)
        {       fputs(*--argv, stderr);
                if (--narg)
                        fputc(' ', stderr);
        }
        fputc('\n', stderr);
}

kind(plp)               /* determine command */
                        /* move line pointer past it and white space */
        int *plp;       /* char**, -> line pointer; NULLed or advanced */
{       char *s;

        for (s = line+1; isspace(*s); ++s)
                ;
        if (*plp = cmd(s, "define"))    return DEFINE;
        if (*plp = cmd(s, "else"))      return ELSE;
        if (*plp = cmd(s, "endif"))     return ENDIF;
        if (*plp = cmd(s, "ifdef"))     return IFDEF;
        if (*plp = cmd(s, "ifndef"))    return IFNDEF;
        if (*plp = cmd(s, "include"))   return INCLUDE;
        if (*plp = cmd(s, "line"))      return LINE;
        if (*plp = cmd(s, "undef"))     return UNDEF;
        return DEFAULT; /* *plp is NULL */
}

cmd(l, c)               /* parse keyword */
                        /* return NULL or -> past it and white space */
        char *l;        /* -> begin of possible keyword */
        char *c;        /* -> keyword */
{
        /* compare */
        while (*l++ == *c++ && *c)
                ;
        if (*c)
                return 0;       /* incomplete keyword */
        if (*l == '\0')
                return 1;       /* just keyword */
        if (! isspace(*l))
                return 0;       /* keyword plus trash */
        while (isspace(*++l))
                ;
        return l;               /* bypassed white space */
}

markfl(sp)              /* bypass and terminate file name */
                        /* return true if found */
        char *sp;       /* -> begin delimeter, " or ( */
```

```
{       char s, *cp, c;


        if ((s = *(cp = sp)) && (s == '\"' || s == '('))
                while (c = *++cp)
                        if (c == '\"' && s == '\"'
                            || c == ')' && s == '(')
                        {       *cp = '\0';
                                return cp-sp ) 1;
                        }
        return 0;
}
```

End Listing One

Listing Two

```
/****
 ****    UN*X compatible dynamic memory allocation
 ****/

/*
 *      calloc  return pointer to vector of 0, or NULL
 *      cfree   free previously allocated area
 *
 *      The heap starts at _end and runs upward toward the stack.
 *      Each area in the heap is preceded by a word at an even address;
 *      a pointer chain runs from _end through these words to NULL;
 *      The low bit in each word is 1 if the following area is free.
 *      There is a blind, allocated element at the front of the chain.
 *
 *      BUG:    very unreasonable demands (e.g., wraparound)
 *              will corrupt memory.
 */

#define SLACK   1024    /* at least 1KB stack to be free */
#define NULL    0

word(wp)
        int *wp;
{
        return *wp;
}

char * calloc(n,len)
        int n;          /* number of elements */
        int len;        /* length of element */
{       int cell;       /* current allocation chain cell */
        char *p;        /* -> cell */
        char *np;       /* pointer in cell */
        int *ip, *wp;   /* for casting */

        len = (len*n + 1) & ~1;         /* even */
        if (len == 0)
                return NULL;
        for (ip = p = word(_end+1 & ~1) & ~1;
             np = (cell = *ip) & ~1;
             ip = p = np)
                if (cell & 1)           /* lowbit == 1 means free */
                {       if ((n = np-p - 2) ) len+2)
                        {       wp = p + len+2;
                                *wp = cell;
```

(Continued on next page)

P (*Listing Continued, text begins on page 46*)

Listing Two

```
                        *ip = wp;
                }
                else if (n >= len)
                        *ip = np;
                else
                        continue;
                for (wp = p+2; len; len -= 2)
                        *wp++ = 0;
                return p+2;
        }
        if ((wp = p + len+2) > &n - SLACK)
                return NULL;
        *ip = wp;
        *wp = NULL;
        for (wp = p+2; len; len -= 2)
                *wp++ = 0;
        return p+2;
}

cfree(fp)
        int *fp;        /* to be freed */
{       int *p, *np;

        --fp;                           /* to cell */
        for (p = _end+1 & ~1;
             np = word(p) & ~1;
             p = np)                    /* p-> previous cell */
                if (np == fp)           /* fp-> cell to free */
                {       np = *fp;       /* np-> following cell */
                        if ((*fp & 1) || np == NULL)
                                break;  /* he does not own it */
                        if (*p & 1)
                                if (*np & 1)
                                        *p = *np;
                                else if (*np == NULL)
                                        *p = NULL;
                                else
                                {       *p = np;
                                        *p |= 1;
                                }
                        else if (*np & 1)
                                *fp = *np;
                        else if (*np == NULL)
                                *fp = NULL;
                        else
                                *fp |= 1;
                        return;
                }
        fputs("cfree botch", stderr);
        exit();
}
```

End Listings

A Simple Minimax Algorithm

Computers frequently use polynomial or rational approximations for transcendental functions, and yet the typical programmer's involvement with these approximations is limited simply to looking up the appropriate parameters of the approximation in a handbook. Have you ever wondered where these approximations came from? Is it easy for programmers with some experience in numerical analysis to invent their own approximations? Suppose you need an approximation that is unavailable in the literature — what do you do?

Possibly these questions have never crossed your mind because you just assumed that these approximations represent a simple Least Squares (LS) fit. I'll tell you right now, these approximations are not a LS fit. Usually they represent a Minimum Maximum Absolute Error (Minimax) fit; that is, rather than determining the approximation parameters so that the mean-squared error or error variance is a minimum over the interval of approximation, the parameters were determined so that the largest absolute value of the errors encountered over the interval of approximation is a minimum. In other words, the worst error has been minimized.

Synonyms for Minimax approximation include "Chebyshev approximation" and "ℓ_∞ approximation." Sometimes, particularly in statistical works, you will see the word "estimate" in place of "approximation." Both LS and Minimax belong to a larger class of approximations that minimize a norm of errors, the particular type of norm defining the type of approximation. Of all the approximations in this class, LS is by far the most popular, most easily calculated, and most discussed.

Because many LS principles carry over to the case of Minimax, this article assumes that the reader has some hands-on experience with LS curve fitting. If you have no such experience and are still interested in Minimax, I suggest that you first acquire some, starting with the vast body of literature on LS.

The reason for choosing Minimax criterion over others such as LS should be obvious. We want to guarantee that the errors associated with an approximation fall within specified limits and to make these limits as small as possible without increasing the amount of time and memory used in calculation. An approximation has two sources of calculation parsimony. First is the form of the approximation. Different forms include odd polynomials, even polynomials, rational functions (quotients of polynomials), and so on. That the selection of form is an art and not a science makes such selection very interesting, but unfortunately that is outside the scope of this article. The second source of parsimony is the type of fit that the approximation uses. For a given form of approximation, the Minimax fit is always the most parsimonious.

In spite of the fact that Minimax approximations are fundamental to computerdom, the techniques for Minimax curve fitting are esoteric, probably because traditional methods are somewhat complicated, often not automatic, and therefore difficult to implement as computer programs. Using conventional techniques, programmers would find it difficult to invent their own approximations. I refer those of you wishing to find out more about these techniques to the work of Hastings,[1] Scheid,[2] and Dem'yanov and Malozemov.[3] The latter provides a rigorous mathematical treatment of the subject and outlines a large number of Minimax algorithms, which, unfortunately, are far removed from the cookbook style that programmers are likely to appreciate. Scheid walks his readers through two of the most popular algorithms, the Exchange Method and a method involving Simplex linear programming, with numerous pencil and paper examples — I highly recommend it. Hastings is the best text for learning the artistic aspects of curve fitting, such as form selection.

To the best of my knowledge, however, this article represents the first publication of actual programs for Minimax curve fitting. I propose to introduce an original Minimax algorithm that offers the advantages of simplicity and accuracy over conventional methods.

With respect to simplicity, this algorithm should result in a minimum of program source code. Some conventional methods, such as the Exchange Method, look deceptively simple until you actually try programming them; the Exchange Method involves data transfers between matrices that are quite messy to program. I challenge anyone to program a conventional Minimax algorithm (with the same degree of functionality) in less than *twice* the source code of the BASIC program of Listing One (page 98).

In using any Minimax algorithm, as you seek higher and higher order approximations, you eventually reach an order where the algorithm fails due to an accumulation of roundoff errors. My algorithm should identify approximations of a higher order (i.e., with more parameters) than any other algorithm using the same precision arithmetic. I have three reasons for believing this:

(1) This algorithm involves the solution of linear equations of order n, whereas other algorithms involve solutions of higher order equations; in the

by Steven A. Ruzinsky

Steven Ruzinsky, 2110 S. Austin Blvd., Cicero, IL 60650

case of the Exchange Method, this order is n + 1 (n is the order of the approximation).
(2) This algorithm solves these linear equations via the Sequential Least Squares algorithm (described later), which is relatively immune to ill-conditioned systems of equations.
(3) To a limited extent, each iteration within the algorithm is compensatory toward roundoff errors incurred from previous iterations, while other algorithms, although iterative, do not have this compensatory effect.

I would be surprised if you can find a Minimax algorithm that will identify higher order approximations than those of the Forth Matrix Language (FML) program of Listing Two (page 99) using double-precision (64-bit) floating point arithmetic. Figures 4–21 (pages 93–97) show the results of the FML program.

Conventions

Before getting into matrix equations, I should define some conventions. Lower-case letters without underscores represent scalar variables. Lower-case letters with underscores represent column vectors. Upper-case letters represent matrices other than vectors. The superscript T indicates matrix transpose. The caret (^) over a variable indicates that it is an approximation or estimate of the variable with the corresponding symbol. The "order" of an approximation is the total number of parameters to be identified, whereas the "degree" of a polynomial is the highest power contained in that polynomial: thus, $y = a_1 x^{10}$ is a polynomial of degree 10 and order 1.

The Error-Fluffing Algorithm

The use of this algorithm is not limited to curve fitting but is applicable to any system that is linear in its parameters. Consider the most general case:

$$y = a_1x_1 + a_2x_2 \cdots + a_nx_n + e \qquad (1)$$

or in matrix notation $y = \underline{a}^T\underline{x} + e$
where

$$\underline{a} \equiv \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \cdot \\ a_n \end{bmatrix}, \quad \underline{x} \equiv \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ x_n \end{bmatrix}, \text{ and } e \equiv \text{Error}$$

which we wish to approximate with

$$\hat{y} = a_1x_1 + a_2x_2 \cdots + a_nx_n \qquad (2)$$

or

$$\hat{y} = \underline{a}^T\underline{x}$$

I anticipate that some people familiar only with the statistical application of LS regression may become confused at this point. Although the error, e, in Equation (1) can be stochastic, in all of the examples used here the error, e, will be completely deterministic. The Minimax algorithm uses LS algorithms, which are derived in the Appendix, without any assumption of the stochasticity or determinability of e.

Given a set of m data points:

$$\{(\underline{x}_i, y_i)\}_m, \; m \geq n,$$

this algorithm iteratively determines a second set of k data points:

$$\{(\underline{x}_i, y_i)\}_k, \; k \geq m,$$

and coefficients $\underline{a}$, such that $\underline{a}$ represents an LS fit to the second set of points and a Minimax fit to the first set of points. In the algorithm's simplest form, $\{(\underline{x}_i, y_i)\}_k$ is a superset of $\{(\underline{x}_i, y_i)\}_m$ containing only multiple replications of the points in $\{(\underline{x}_i, y_i)\}_m$
A good way to conceptualize this is to consider the first set of points as being locations $(\underline{x},y)$ space and the second set as having points at those locations, with many of the locations having multiple points from the second set.

The procedure, simplified to facilitate understanding, is:
Let k = iteration index, and
(1) Use an LS fit as the initial estimate of $\underline{a}_0$.
(2) Find the maximum absolute error resulting from $\underline{a}_k$.
(3) Reincorporate the point at which the maximum absolute error occurs back into the data set.
(4) Perform LS regression on the new data set.
(5) Repeat, starting at Step 2, until no further decrease in maximum absolute error seems likely with further iteration (convergence is nonmonotone).

Since a rigorous mathematical convergence proof of this algorithm is presently unknown, I cannot guarantee that it will always work. However, except for failures due to roundoff errors, I have yet to encounter a single case where this algorithm didn't work! In a practical sense, this Minimax algorithm is more foolproof than LS algorithms because we can easily check to see if the results are truly Minimax. The error curve (i.e., e versus x) of a Minimax fit will contain at least n + 1 minima and maxima that are equal in absolute value; that is, at least n + 1 equal worst errors will appear. Remember, n is the number of estimated parameters and not necessarily the degree of a polynomial. In almost all cases of practical interest, the error curve will have exactly n + 1 equal and unique minima and maxima; under this condition, we can be reasonably certain that the fit is Minimax. By contrast, LS algorithms, which have a firm mathematical foundation, frequently generate erroneous results (because of roundoff errors) that go undetected — there is no easy way to check a LS result.

Although I lack a convergence proof of the Minimax algorithm, you should be able to understand at an *intuitive* level how the algorithm works. Replicating the data point at which the maximum absolute error occurs gives increased weight to the location of that point; that is, to minimize the error variance with respect to the new set of points, the error at that location will be reduced while increasing elsewhere. As this process repeats, the maximum absolute error eventually appears at a new location. This new location receives the same treatment, but now previously weighted locations will resist error increases. Because this constrains the maximum absolute error, it will nonmonotonically approach a minimum. Incidentally, it is this nonmonotone nature of the algorithm that makes a mathematical convergence proof difficult.

It is very interesting to watch the change in the error curve with each iteration. Because the process resembles fluffing the bulges out of a pillow, I use the term "error fluffing" to describe this algorithm.

The SLS Algorithm

You should have noticed two impractical aspects to the five-step simplified procedure. First, with every iteration,

Step 3 adds one point to the data set. Since thousands of iterations may be required, this growing set of data points could easily exceed the memory capacity of even the largest mainframe! Second, in Step 4, performing LS regression on even a moderately large set of points is very time-consuming.

Fortunately, we can combine Steps 3 and 4 so that the set $\{(\underline{x}_i, y_i)\}_k$ exists only as a mathematical abstraction without occupying any RAM; remember, we are only interested in getting $\underline{a}_k$, the coefficients corresponding to an LS fit to $\{(\underline{x}_i, y_i)\}_k$

The method for doing this, which is also very fast, is called the Sequential Least Squares (SLS) algorithm. Indeed, without the SLS algorithm, this Minimax algorithm would be of no practical value. Using the SLS algorithm, the time per iteration for this Minimax algorithm is significantly less than that of other iterative algorithms such as the Exchange Method. Thus, even though such algorithms may converge in less iterations, my algorithm may be as fast!

I would like to digress a moment and say that the SLS algorithm is the most wasted mathematical resource I know. Outside of the electrical engineering community, this algorithm is almost unknown. For example, of the numerous statistical software packages on the market, to the best of my knowledge none uses the SLS algorithm. The SLS algorithm, which resembles a Kalman filter, was once used for real-time analysis of time series (e.g., in speech recognition), but recently the faster "lattice" or "ladder" algorithms have replaced it. Use of these newer algorithms, however, is limited to identifying ARMA (AutoRegressive-Moving Average) parameters for time series, whereas the SLS algorithm is applicable to *all* forms of linear regression.

The SLS algorithm is also particularly well suited for a large variety of microcomputer applications because it conserves memory and is, without any doubt, the easiest LS algorithm to program. The SLS algorithm avoids matrix inversion and involves little more than matrix multiplication and addition. By letting m = n, you can even use the SLS algorithm in place of such algorithms as the Gaussian Elimination to solve n linear equations in n unknowns or to invert a matrix! I have included a complete description and derivation of the SLS algorithm in the Appendix. For further information on LS algorithms and their applications to time series analysis, see the two books by Graupe.[4,5]

Implementation

The two implementations of the Minimax algorithm include one in BASIC (Listing One) and another in Polyforth with FML (Listing Two). FML is an APL-like extension to Forth developed by this author, which, when used with the 8087, I believe to be the fastest numerical language available for microcomputers. Version 1.0 of FML is in the public domain and can be found in Nos. 80, 81, and 82 of *Dr. Dobb's Journal*. The illustrative program presented here is written in the commercially available version 3.0. Besides speed, the main advantage of FML is that it performs matrix operations with syntactically simple statements, thus facilitating quick and convenient "on the fly" programming of complex mathematical algorithms; for example, LS regression is performed by a single word in version 3.0. FML also allows arrays of up to 256Kbytes, thus allowing storage of huge data sets.

Such being the case, some drastic differences exist between the FML and BASIC implementations. Because the BASIC program represents the algorithm in its simplest possible form, it will not be able to duplicate the performance of the FML program in regard to either speed or accuracy. The BASIC program is described in the comment fields, and major differences between the BASIC and FML programs are mentioned in the text describing the FML program. If anyone is interested in the nitty-gritty details of how the BASIC program works, I suggest that you first study the FML program description, then the SLS section of the Appendix, and last the comment field of the BASIC program.

The BASIC program employs an example used by both Hastings[1] (page 138) and Ruckdeschel[6] (page 522). Here Equation (1) takes the form:

$$y = a_1x + a_2x^3 + a_3x^5 + e$$

where

$$y = \text{SIN}(PI*X/2)$$

However, in this particular instance, $\underline{a}$ is determined to a Minimum Maximum Absolute *Relative* Error criterion. Dividing both sides of the above equation by y:

$$(1) = a_1(x/y) + a_2(x^3/y) + a_3(x^5/y) + (e/y)$$

and redefining y, $\underline{x}$, and e to be the bracketed quantities puts Equation (1) in a form so that $\underline{a}$ will be determined in a way to minimize the relative error, (e/y). The coefficients resulting from this program are $a_1 = 1.5706258$, $a_2 = -0.6432224$, and $a_3 = 0.0727045$; the resulting maximum absolute relative error is 0.00010856. Hastings' result is $a_1 = 1.5706268$, $a_2 = -0.6432292$, and $a_3 = 0.0727102$, with a maximum absolute relative error of 0.00010879.

Figure 1 (page 92) shows the error curve for a Minimax fit, whereas Figure 2 (page 92) shows the curve for an LS fit. The similarities and differences between these two error curves are typical of LS and Minimax fits in general. Notice that the worst error of the LS fit occurs near the range ends and that the worst error of the Minimax fit is 55% smaller. Ruckdeschel attempted a Minimax fit using a very general optimization by steepest descent algorithm; Figure 3 (page 92) shows the resulting error curve. Clearly, his fit is not Minimax, thereby demonstrating the inadequacy of gradient algorithms for this purpose.

You can easily modify the BASIC program to find other approximations. For example, setting N = 4 in line 140 will result in a fourth order approximation.

The FML Implementation

Screen 207 contains the final FML program, formatted as a list of steps. I do not expect you to be able to read FML. A detailed description of the steps follows, so that no one should have much difficulty in implementing the algorithm in the language of their choice.

Insert Data

First, the set of sample points

$$\{(\underline{x}_i, y_i)\}_k$$

is entered into matrices and X as follows:

$$\underline{y} \equiv \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ y_m \end{bmatrix}$$

and

$$X \equiv \begin{bmatrix} \underline{x}_1^T \\ \underline{x}_2^T \\ \underline{x}_3^T \\ . \\ \underline{x}_m^T \end{bmatrix} \equiv \begin{bmatrix} x_{11}\, x_{12}\, x_{13} \cdots x_{1n} \\ x_{21}\, x_{22}\, x_{23} \cdots x_{2n} \\ x_{31}\, x_{32}\, x_{33} \cdots x_{3n} \\ . \quad . \quad . \quad \cdots \\ x_{m1} x_{m2} x_{m3} \cdots x_{mn} \end{bmatrix}$$

You should realize that the Minimax algorithm is applicable only to a finite number of points and not to the actual function, which is continuous. Because of this, we must hope that the resulting approximation will accurately interpolate between the sample points of the function being approximated. Generally, inspection of the error curve and the use of common sense are all that is needed to verify that the number of points, m, is sufficient.

As a rule, approximations with larger numbers of parameters will also require more points because the peaks and valleys in the error curve will be sharper and therefore more likely to protrude between sample points. If these peaks and valleys are crowded (hence, more acute at one or both ends of the interval of approximation), you then can select samples of x that are crowded at the ends. Note that in this respect Minimax is totally different from LS. To achieve an LS approximation to a continuous function, you generally must select points at equal intervals of x.

Batch Least Squares

Next, the LS coefficients, $\underline{a}_o$, are calculated:

$$Q = (X^T X)^{-1} \qquad (3)$$

$$\underline{a}_0 = Q(X^T \underline{y}) \qquad (4)$$

Q, an n x n matrix, is called the "inverse covariance matrix." The parentheses in Equation (4) indicate the preferred order of calculation. Actually, you can substitute almost any LS algorithm, including those that do not use matrix inversion, for Equation (4), but it is necessary to calculate Q, Equation (3), because Q will be used later. (The BASIC program uses the SLS algorithm in place of Equations (3) and (4). Using SLS from a cold start introduces an error into the calculations, as explained in the Appendix.)

Reduce Roundoff Error

Large roundoff errors often result from LS algorithms such as those of Equations (3) and (4). Ruckdeschel describes a method to reduce these errors. First, the error vector, e, is calculated:

$$\underline{e} \equiv \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ . \\ e_m \end{bmatrix} = \underline{y} - X\underline{a} \qquad (5)$$

If we redesignate the old coefficient vector as, a′, the improved estimate, is:

$$\underline{a} = \underline{a}' + Q(X^T \underline{e}) \qquad (6)$$

Equations (5) and (6) should be repeated iteratively, and the best $\underline{a}$ (i.e., the $\underline{a}$ corresponding to the smallest variance) saved. Equation (6) is tantamount to LS regression on the error, $\underline{e}$, adding the resulting coefficients to $\underline{a}$. Again, you can substitute almost any other LS algorithm in this step. (The BASIC program skips this step.)

Normalize Q

The Q matrix is used in succeeding calculations. Although the Minimax algorithm will work with Q as is, increasing the values in Q by a constant factor will hasten convergence considerably. Beyond a certain limit, however, convergence may be lost completely. I found a good scaling factor to be m/10n. If we redesignate the old matrix as Q′, the new matrix will be:

$$Q = (m/10n)Q'$$

(The BASIC program skips this step.)

The following operations are reiterated, k being the iteration index; ultimately, $\underline{a}$, will converge to a Minimax fit.

Find MAX(e), xk; Record Best a, ek

First, Equation (5) is used to calculate $\underline{e}$, then the maximum absolute error, $|e_k|$ is found, and e_k and the corresponding $\underline{x}_k$ from the matrix X are recorded for future use. Since convergence is nonmonotone, the best $|e_k|$ among all iterations to this point in time and the corresponding $\underline{a}_k$ should be recorded. After iteration is terminated, this best $\underline{a}_k$ is used as the final result.

Sequential Least Squares

As previously mentioned, the SLS algorithm is the key to making this Minimax algorithm practical. The SLS algorithm adds a data point by updating the Q matrix and $\underline{a}_k$ without updating X and $\underline{y}$. Thus, X and $\underline{y}$ do not grow in dimension, and we can avoid the time-consuming operations represented by Equations (3), (4), and (6):

$$Q_k = Q_{k-1} - \frac{Q_{k-1}\, \underline{x}_k\, \underline{x}_k^T\, Q_{k-1}}{1 + \underline{x}_k^T\, Q_{k-1}\, \underline{x}_k} \qquad (8a)$$

$$\underline{a}_k = \underline{a}_{k-1} + Q_k\, \underline{x}_k\, (y_k - \underline{x}_k^T\, \underline{a}_{k-1}) \qquad (8b)$$

or: $\underline{a}_k = \underline{a}_{k-1} + Q_k\, \underline{x}_k\, e_k$

Notice that only in the context of Minimax do ($\underline{x}_k, y_k$), and e_k represent the data point and error corresponding to $|e_k|$ the maximum absolute error that results from $\underline{a}_{k-1}$.

Speedup

The SLS algorithm is fast and contributes negligible time to the Minimax algorithm. The bottleneck here is in the calculation of $\underline{e}$ with each iteration. After a certain number of iterations, however, we may use the fact that the maximum absolute errors now appear at only a small subset of the original sample points to hasten the process. The method used here consists of sorting the rows of $\underline{y}$ and X with respect to a descending order of absolute values of the elements of $\underline{e}$ then halving the dimensions, m, of, $\underline{y}$, $\underline{e}$, and X by truncation. (The BASIC program skips this step.)

The Results

Now that we've finished describing the Minimax algorithm in FML, let's look at some results. Figures 4–21 show the parameters and resulting error curves of approximations to nine different functions. All of these approximations were found with the FML program. The even-numbered figures duplicate

(with greater accuracy) the highest order results of Hastings for the indicated functions. The odd-numbered figures represent the highest order fits that I was able to achieve for those same functions. In many instances, the program identified more than twice Hastings' number of parameters. The table (at left) compares the accuracy of various approximations.

You should notice that the approximations given in Figures 10, 11, 12, 13, 18, 19, 20, and 21 are not linear in their parameters. How, then, did I manage to use my algorithm on them? There are a number of tricks for adapting linear algorithms to nonlinear situations. In the above cases, a simple transformation changes these approximations into the form of Equation (1) so that the algorithm can be successfully applied. As a contest, a free one-year subscription to *Dr. Dobb's* will be awarded to the first person to correctly describe this transformation.

Now to answer the question: "How fast is this Minimax algorithm?" A suitable reference for comparison is the time taken by a Batch LS algorithm to fit the same data. For a case with n = 12 and m = 320, the Batch LS portion of the FML program takes 2.37 sec, whereas each iteration of the loop of the Minimax algorithm takes 0.53 sec. (before SPEEDUP). As previously mentioned, the time contribution of the SLS algorithm, 0.08 sec per iteration, is negligible. The calculation of the $\underline{e}$ vector accounts for most of the 0.53 seconds.

Initial convergence is rapid, so that only 200–500 iterations may be required to achieve the accuracy obtained by Hastings. However, maximum accuracy is achieved at about 40,000 iterations; with double precision, roundoff errors prevent gains in accuracy with further iteration.

In summary, the Minimax algorithm presented here is simple enough for those interested in numerical analysis to use on a casual basis. With this algorithm and double-precision arithmetic, programmers can find Minimax approximations that are both of a higher order and more accurate than those published in the literature. Furthermore, the use of this algorithm is not limited to deterministic systems.

It may be interesting to explore the use of Minimax in statistics. Typically, LS is used because, for systems in which the error is random and normally distributed, LS either is or closely approximates a maximum likelihood estimator of the parameters. For certain systems where the error has a statistical distribution that is bounded, however, Minimax may be more statistically efficient.

FUNCTION	FIGURE	NUMBER OF PARAMETERS	MAXIMUM ABSOLUTE ERROR	PERCENT IMPROVEMENT OVER HASTINGS' RESULT	PERCENT IMPROVEMENT OVER LEAST SQUARES
$LOG_{10}(X)$	4	5	1.32254 E-07	0.8	56
	5	12	1.11137 E-15		62
ARCTAN(X)	6	8	3.75120 E-08	0.7	63
	7	12	2.25114 E-11		42
$SIN\left(\frac{\pi}{2}x\right)$	1	3	1.08179 E-04*	0.6	55
	8	5	5.31748 E-09*	0.9	62
	9	7	6.28159 E-14*		66
10^x	10	7	4.96196 E-09*	0.9	
	11	10	1.18194 E-13*		
$\frac{1}{\sqrt{2\pi}}e^{-\frac{x}{2}^2}$	12	6	2.21756 E-04	0.1	
	13	16	2.32245 E-09		
ARCSIN(X)	14	8	2.18036 E-08	0.6	62
	15	13	1.60434 E-12		67
LN(1+X)	16	8	3.21271 E-08	0.3	52
	17	13	2.99006 E-12		61
e^{-x}	18	6	2.42142 E-07	0.1	
	19	12	3.42650 E-13		
ERF(X)	20	6	2.59522 E-07	0.3	
	21	14	1.41491 E-12		

*RELATIVE ERROR

Table

Comparison of Minimax Results with that of Hastings and LS

References

1. Hastings, Cecil, *Approximations for Digital Computers*, Princeton University Press, 1955.

2. Scheid, F., *Numerical Analysis, Schaum's Outline Series*, McGraw-Hill, 1968.

3. Dem'yanov, V. F., and Malozemov, V. N., *Introduction to Minimax*, John Wiley & Sons, 1974.

4. Graupe, D., *Identification of Systems*, Krieger, 1972.

5. Graupe, D., *Time Series Analysis: Identification and Adaptive Filtering*, Krieger, 1983.

6. Ruckdeschel, F. F., *Basic Scientific Subroutines*, Vol. 2, BYTE Publications, Inc., 1981.

Suggested Additional Reading

1. Rivlin, T., *An Introduction to the Approximation of Functions*, Dover, 1969.

2. Davis, P. J., *Interpolation and Approximation*, Dover, 1975.

Appendix: Derivation of Least Squares Algorithms

Much of the material that follows is based on the literature. Readers may consult the references accompanying this article as a starting place for additional material on the subject. I use nomenclature from the text.

Batch Least Squares

Let MSE ≡ Sample Mean Squared Error

then $MSE = 1/k \sum_{i=1}^{k} e_i^2$

or $MSE = 1/k\ (\underline{e}^T\underline{e})$

Let tr ≡ trace operator ≡ sum of diagonal elements of a square matrix.

since $\underline{e}^T\underline{e} = tr(\underline{e}\underline{e}^T)$

and $\underline{e} = \underline{y} - X\underline{a}$

then $MSE = 1/k\ tr[(\underline{y} - X\underline{a})\ (\underline{y} - X\underline{a})^T]$

$MSE = 1/k\ tr[(\underline{y} - X\underline{a})(\underline{y}^T - \underline{a}^T x^T)]$

$MSE = 1/k\ tr[\underline{y}\underline{y}^T + X\underline{a}\underline{a}^T X^T - X\underline{a}\underline{y}^T - \underline{y}\underline{a}^T X^T]$

$MSE = 1/k[tr(\underline{y}\underline{y}^T) + tr(X\underline{a}\underline{a}^T X^T) - tr(X\underline{a}\underline{y}^T) - tr(\underline{y}\underline{a}^T X^T)]$

The gradient of a scaler, c, with respect to a matrix, Z, is defined as:

$$\frac{\partial c}{\partial Z} \equiv \begin{bmatrix} \frac{\partial c}{\partial z_{11}} & \frac{\partial c}{\partial z_{12}} & \cdots & \frac{\partial c}{\partial z_{1n}} \\ \frac{\partial c}{\partial z_{21}} & \frac{\partial c}{\partial z_{22}} & \cdots & \frac{\partial c}{\partial z_{2n}} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \frac{\partial c}{\partial z_{m1}} & \frac{\partial c}{\partial z_{m2}} & \cdots & \frac{\partial c}{\partial z_{mn}} \end{bmatrix}$$

Graupe [4] gives rules for the gradation of trace functions. Those of relevance here are:

$$\frac{\partial}{\partial Z} tr(A) = [0]$$

$$\frac{\partial}{\partial Z} tr(A Z Z^T B) = (A^T B^T + B A) Z$$

$$\frac{\partial}{\partial Z} tr(A Z B) = A^T B^T$$

Since the MSE is a positive hyperparabolic function of $\underline{a}$, a sufficient condition for minimizing the MSE is:

$$\frac{\partial MSE}{\partial a_i} = 0 \text{ for all } i$$

$$\frac{\partial}{\partial \underline{a}} MSE = \underline{0}$$

therefore

$$\frac{\partial}{\partial \underline{a}} tr(\underline{y}\underline{y}^T) + \frac{\partial}{\partial \underline{a}} tr(X\underline{a}\underline{a}^T X^T) - \frac{\partial}{\partial \underline{a}} tr(X\underline{a}\underline{y}^T) - \frac{\partial}{\partial \underline{a}} tr(\underline{y}\underline{a}^T X^T) = \underline{0}$$

Applying the rules for gradation of trace functions:

$$\frac{\partial}{\partial \underline{a}} tr(\underline{y}\underline{y}^T) = \underline{0}$$

$$\frac{\partial}{\partial \underline{a}} tr(X\underline{a}\underline{a}^T X^T) = 2X^T X\underline{a}$$

$$\frac{\partial}{\partial \underline{a}} tr(X\underline{a}\underline{y}^T) = X^T\underline{y}$$

$$\frac{\partial}{\partial \underline{a}} tr(\underline{y}\underline{a}^T X^T) = \frac{\partial}{\partial \underline{a}} tr(X\underline{a}\underline{y}^T) = X^T\underline{y}$$

thus

$$\underline{0} + 2X^T X\underline{a} - X^T\underline{y} - X^T\underline{y} = \underline{0}$$

$$X^T X\underline{a} - X^T\underline{y} = \underline{0}$$

$$\underline{a} = (X^T X)^{-1} X^T\underline{y}$$

Sequential Least Squares

This algorithm provides an efficient means of performing regression on a growing data set. It does this by directly updating Q and $\underline{a}$, one data point at a time, without actually recording the data. However, to show mathematical equivalency to the Batch Least Squares algorithm above, we must imagine a growing $\underline{y}$, and X:

$$\underline{y}_k = \begin{bmatrix} \underline{y}_{k-1} \\ y_k \end{bmatrix} \text{ and } X_k = \begin{bmatrix} X_{k-1} \\ \underline{x}_k^T \end{bmatrix}$$

Observe that $\underline{y}_{k-1}$ and x_{k-1} represent old data and $(\underline{x}_k, y_k)$ is a new data point that is being added to form the new data set represented by $\underline{y}_k$ and X_k. Then starting with the previously derived Batch Least Squares:

$$\underline{a}_k = (X_k^T X_k)^{-1} X_k^T\underline{y}_k$$

$$Q_k = (X_k^T X_k)^{-1}$$

$$Q_k^{-1} = X_k^T X_k$$

$$Q_k^{-1} = \sum_{i=1}^{k} \underline{x}_i\underline{x}_i^T$$

$$Q_k^{-1} = \sum_{i=1}^{k-1} \underline{x}_i\underline{x}_i^T + \underline{x}_k\underline{x}_k^T$$

$$Q_k^{-1} = Q_{k-1}^{-1} + \underline{x}_i\underline{x}_i^T$$

Next, the Matrix Inversion Lemma will be derived and used. Consider any nonsingular square matrices A and C and conformable matrices B and D:

$$D + DA^{-1}BCD = D + DA^{-1}BCD$$

$$DA^{-1}(A + BCD) = (C^{-1} + DA^{-1}B)CD$$

$$(C^{-1} + DA^{-1}B)^{-1}DA^{-1}(A + BCD) = CD$$

$$A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1}(A + BCD) = A^{-1}BCD$$

$$I + A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1}(A + BCD) = I + A^{-1}BCD$$

$$I + A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1}(A + BCD) = A^{-1}(A + BCD)$$

$$(A + BCD)^{-1} + A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1} = A^{-1}$$

$$(A + BCD)^{-1} = A^{-1} - A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1}$$

Since $Q_k = (Q_{k-1}^{-1} + \underline{x}_k\underline{x}_k^T)^{-1}$

then by the Matrix Inversion Lemma:

$$Q_k = Q_{k-1} - Q_{k-1}\underline{x}_k(1 + \underline{x}_k^T Q_{k-1}\underline{x}_k)^{-1}\underline{x}_k^T Q_{k-1}$$

$$Q_k = Q_{k-1} - \frac{Q_{k-1}\,\underline{x}_k\,\underline{x}_k^T\,Q_{k-1}}{1 + \underline{x}_k^T\,Q_{k-1}\,\underline{x}_k}$$

$$\underline{a}_k = Q_k X_k^T\underline{y}_k$$

$$X_k^T\underline{y}_k = \sum_{i=1}^{k} \underline{x}_i y_i$$

$$X_k^T\underline{y}_k = \sum_{i=1}^{k-1} \underline{x}_i y_i + \underline{x}_k y_k$$

$$X_k^T\underline{y}_k = X_{k-1}^T\underline{y}_{k-1} + \underline{x}_k y_k$$

$$X_k^T\underline{y}_k = Q_{k-1}^{-1}Q_{k-1}X_{k-1}^T\underline{y}_{k-1} + \underline{x}_k y_k$$

$$\underline{a}_{k-1} = Q_{k-1}X_{k-1}^T\underline{y}_{k-1}$$

$$X_k^T\underline{y}_k = Q_{k-1}^{-1}\underline{a}_{k-1} + \underline{x}_k y_k$$

$$Q_{k-1}^{-1} = Q_k^{-1} - \underline{x}_k\underline{x}_k^T$$

$$X_k^T\underline{y}_k = (Q_k^{-1} - \underline{x}_k\underline{x}_k^T)\,\underline{a}_{k-1} + \underline{x}_k y_k$$

$$\underline{a}_k = Q_k X_k^T\underline{y}_k$$

$$\underline{a}_k = Q_k[(Q_k^{-1} - \underline{x}_k\underline{x}_k^T)\,\underline{a}_{k-1} + \underline{x}_k y_k]$$

$$\underline{a}_k = (Q_k Q_k^{-1} - Q_k\underline{x}_k\underline{x}_k^T)\,\underline{a}_{k-1} + Q_k\underline{x}_k y_k$$

$$\underline{a}_k = (I - Q_k\underline{x}_k\underline{x}_k^T)\,\underline{a}_{k-1} + Q_k\underline{x}_k y_k$$

$$\underline{a}_k = \underline{a}_{k-1} + Q_k\underline{x}_k\,(y_k - \underline{x}_k^T\underline{a}_{k-1})$$

If the Sequential Least Squares algorithm is to be used from a cold start, the initial values, $\underline{a}_0 = \underline{0}$, and Qo = cI should be used where c is the largest number possible without causing failure of the algorithm due to roundoff errors. Ideally, Qo^{-1} should equal [0] but, since this is singular, an ideal Qo does not exist. Hence, for any finite Qo, an error is introduced

(Continued on page 97)

```
Minimax Fit
Function:  SIN(X*PI/2)      Range: 0.0 to 1.0
Maximum Absolute Relative
  Error = 1.08178979E-04
Error Variance = 5.82237977E-09
Coefficients:
A(0) =  1.5706264010388677E+00
A(1) = -6.4322566310483564E-01
A(2) =  7.2707440066142992E-02
```

$$\text{SIN}\left(\frac{\pi}{2}x\right) \approx a_0x + a_1x^3 + a_2x^5,\ -1 \leqslant x \leqslant 1$$

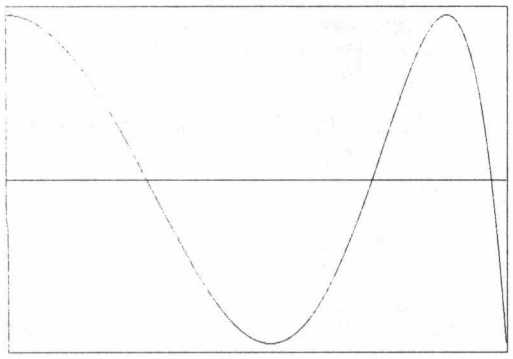


Figure 1

The Relative Error Curve Resulting from the Minimax Fitting of a 5th Degree (n=3) Polynomial to SIN(PI*X/2)

```
Least Squares Fit
Function:  SIN(X*PI/2)      Range: 0.0 to 1.0
Maximum Absolute Relative
  Error = 2.40870170E-04
Error Variance = 4.50629649E-09
Coefficients:
A(0) =  1.5706763023886539E+00
A(1) = -6.4370505986273774E-01
A(2) =  7.3269627643904808E-02
```

$$\text{SIN}\left(\frac{\pi}{2}x\right) \approx a_0x + a_1x^3 + a_2x^5,\ -1 \leqslant x \leqslant 1$$

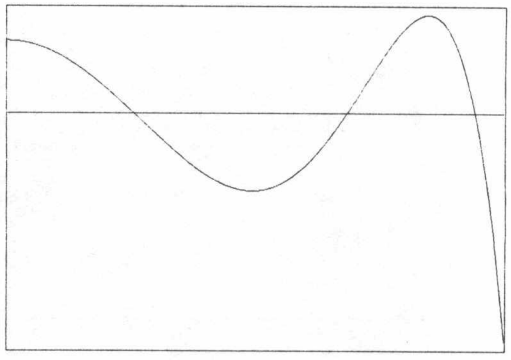


Figure 2

The Relative Error Curve Resulting from a Least Squares Fit of a 5th Degree (n=3) Polynomial to SIN(PI*X/2)

```
 Ruckdeschel:
Minimax Fit
Function:  SIN(X*PI/2)      Range: 0.0 to 1.0
Maximum Absolute Relative
  Error = 1.43752885E-04
Error Variance = 8.49330500E-09
Coefficients:
A(0) =  1.5706894000000000E+00
A(1) = -6.4323300000000005E-01
A(2) =  7.2555999999999996E-02
```

$$\text{SIN}\left(\frac{\pi}{2}x\right) \approx a_0x + a_1x^3 + a_2x^5,\ -1 \leqslant x \leqslant 1$$

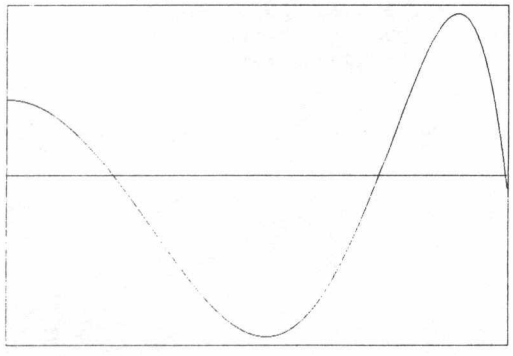


Figure 3

The Relative Error Curve Resulting from Ruckdeschel's Attempt to Achieve a Minimax Fit Using Optimization by Steepest Descent to SIN(PI*X/s)

Figures 4 through 21

Figures 4 through 21 represent approximations found by the minimax algorithm implemented in the Forth Matrix Language program of Listing 2. The even numbered figures duplicate the highest ordered approximations found by Hastings whereas the odd numbered figures represent approximations of a significantly higher order than those of Hastings.

```
Minimax Fit
Function: LOG(X)   Range: 1.0 to SQRT(10.0)
Maximum Absolute Error = 1.32254277E-07
Error Variance = 8.71559890E-15
Coefficients:
A(0) =  8.6859170033741495E-01
A(1) =  2.8933610841477914E-01
A(2) =  1.7751576585077944E-01
A(3) =  9.4403559884429847E-02
A(4) =  1.9129734604173343E-01
```

$$\mathrm{LOG}_{10}(x) \approx a_0\left(\frac{x-1}{x+1}\right) + a_1\left(\frac{x-1}{x+1}\right)^3 + \cdots + a_4\left(\frac{x-1}{x+1}\right)^9,\ \frac{1}{\sqrt{10}} \leqslant x \leqslant \sqrt{10}$$

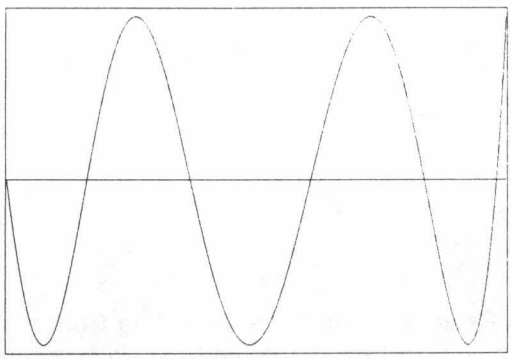


Figure 4

```
Minimax Fit
Function: LOG(X)   Range: 1.0 to SQRT(10.0)
Maximum Absolute Error = 1.11137499E-15
Error Variance = 5.76209886E-31
Coefficients:
A(0) =  8.6858896380645279E-01
A(1) =  2.8952965462124397E-01
A(2) =  1.7371779062864265E-01
A(3) =  1.2408424804953024E-01
A(4) =  9.6506696899528205E-02
A(5) =  7.9019310730381306E-02
A(6) =  6.6160376582026767E-02
A(7) =  6.2932880597822538E-02
A(8) =  2.5321022483988299E-02
A(9) =  1.3171606514518003E-01
A(10) = -1.3290449657183437E-01
A(11) =  2.1590497542362960E-01
```

$$\mathrm{LOG}_{10}(x) \approx a_0\left(\frac{x-1}{x+1}\right) + a_1\left(\frac{x-1}{x+1}\right)^3 + \cdots + a_{11}\left(\frac{x-1}{x+1}\right)^{23},\ \frac{1}{\sqrt{10}} \leqslant x \leqslant \sqrt{10}$$

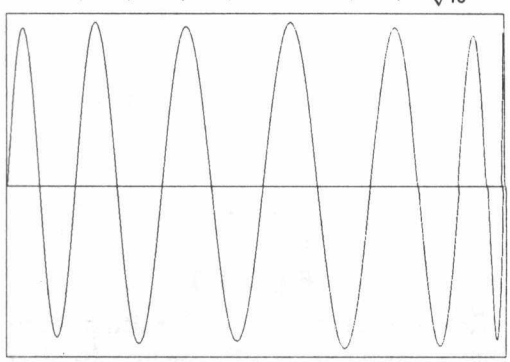


Figure 5

```
Minimax Fit
Function:  ARCTAN(X)   Range: 0.0 to 1.0
Maximum Absolute Error = 3.75119864E-08
Error Variance = 7.01558800E-16
Coefficients:
A(0) =  9.9999933573499511E-01
A(1) = -3.3329861463931842E-01
A(2) =  1.9946573483723884E-01
A(3) = -1.3908668702150279E-01
A(4) =  9.6422971343165645E-02
A(5) = -5.5913682046468495E-02
A(6) =  2.1863890550873954E-02
A(7) = -4.0548228214270449E-03
```

$$\mathrm{ARCTAN}(x) \approx a_0x + a_1x^3 + \cdots + a_7x^{15},\ -1 \leqslant x \leqslant 1$$

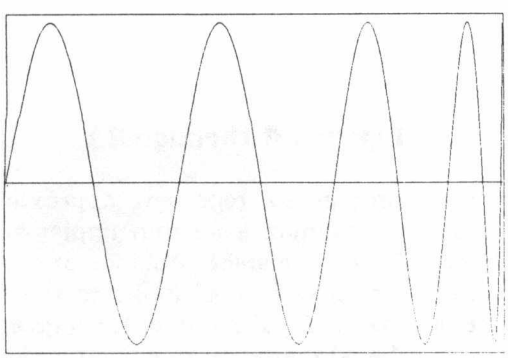


Figure 6

```
Minimax Fit
Function:  ARCTAN(X)   Range: 0.0 to 1.0
Maximum Absolute Error = 2.25113635E-11
Error Variance = 2.44028609E-22
Coefficients:
A(0) =  9.9999999943085005E-01
A(1) = -3.3333327048062561E-01
A(2) =  1.9999793768255469E-01
A(3) = -1.4282554529129779E-01
A(4) =  1.1083694502163881E-01
A(5) = -8.9412233101935801E-02
A(6) =  7.1433729243175675E-02
A(7) = -5.2520067625352863E-02
A(8) =  3.2239154415053844E-02
A(9) = -1.4726129937627916E-02
A(10) =  4.2957241092664187E-03
A(11) = -5.8808009033101821E-04
```

$$\mathrm{ARCTAN}(x) \approx a_0x + a_1x^3 + \cdots + a_{11}x^{23},\ -1 \leqslant x \leqslant 1$$

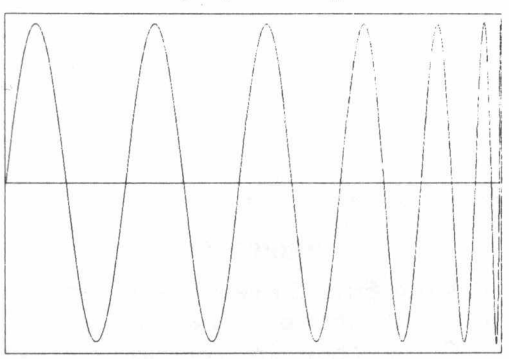


Figure 7

```
Minimax Fit
Function:  SIN(X*PI/2)      Range: 0.0 to 1.0
Maximum Absolute Relative
  Error = 5.31747856E-09
Error Variance = 1.41003464E-17
Coefficients:
A(0) =  1.5707963184491878E+00
A(1) = -6.4596371064188951E-01
A(2) =  7.9689679172657638E-02
A(3) = -4.6737669936775450E-03
A(4) =  1.5148532647329515E-04
```

$$\text{SIN}\left(\frac{\pi}{2}x\right) \approx a_0x + a_1x^3 + \cdots + a_4x^9,\ -1 \leqslant x \leqslant 1$$

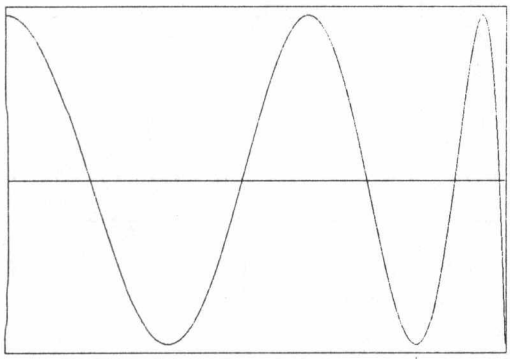


Figure 8

```
Minimax Fit
Function:  SIN(X*PI/2)      Range: 0.0 to 1.0
Maximum Absolute Relative
  Error = 6.28158853E-14
Error Variance = 1.94727887E-27
Coefficients:
A(0) =  1.5707963267947991E+00
A(1) = -6.4596409749718697E-01
A(2) =  7.9692626107052855E-02
A(3) = -4.6817533267793027E-03
A(4) =  1.6043893116473893E-04
A(5) = -3.5955991943598100E-06
A(6) =  5.4590205930680611E-08
```

$$\text{SIN}\left(\frac{\pi}{2}x\right) \approx a_0x + a_1x^3 + \cdots + a_6x^{13},\ -1 \leqslant x \leqslant 1$$

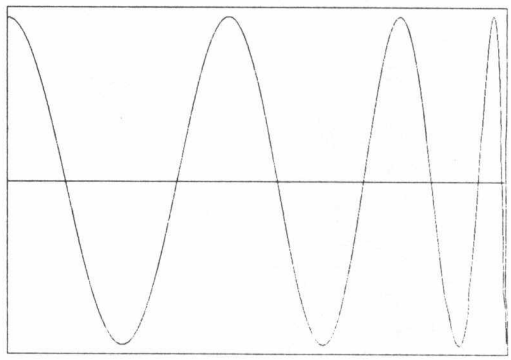


Figure 9

```
Minimax Fit
Function: 10^X Range: 0.0 to 1.0
Maximum Absolute Relative Error = 4.96196032E-09
Error Variance = 1.22996227E-17
Coefficients:
A(0) =  1.1512927746664381E+00
A(1) =  6.6273090331202300E-01
A(2) =  2.5439346793918238E-01
A(3) =  7.2952042631990802E-02
A(4) =  1.7420655137681311E-02
A(5) =  2.5552719905680065E-03
A(6) =  9.3253665510615105E-04
```

$$10^x \approx (1 + a_0x + a_1x^2 + \cdots + a_6x^7)^2,\ 0 \leqslant x \leqslant 1$$

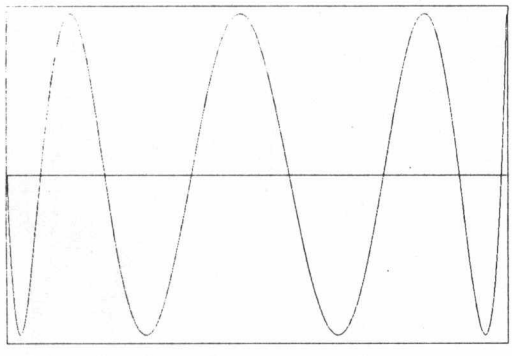


Figure 10

```
Minimax Fit
Function: 10^X Range: 0.0 to 1.0
Maximum Absolute Relative Error = 1.18194099E-13
Error Variance = 6.91798176E-27
Coefficients:
A(0) =  1.1512925464870498E+00
A(1) =  6.6273726433125657E-01
A(2) =  2.5433481471199015E-01
A(3) =  7.3203528479750382E-02
A(4) =  1.6855309537847786E-02
A(5) =  3.2355525426469244E-03
A(6) =  5.2954928235629073E-04
A(7) =  7.9479944032521686E-05
A(8) =  7.6291259328835485E-06
A(9) =  1.9857253301916773E-06
```

$$10^x \approx (1 + a_0x + a_1x^2 + \cdots + a_9x^{10})^2,\ 0 \leqslant x \leqslant 1$$

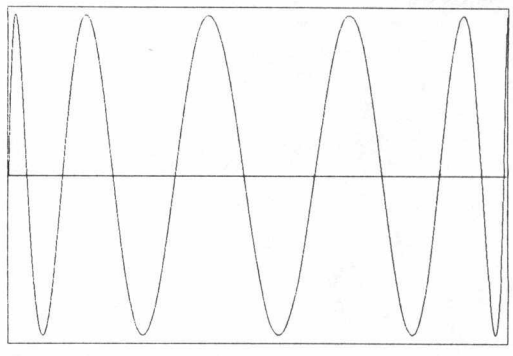


Figure 11

```
Minimax Fit
Function: (1/SQRT(2*PI))*EXP(-X*X/2)
  Range: 0.0 to 5.0
Maximum Absolute Error = 2.21755841E-04
Error Variance = 2.27665975E-08
Coefficients:
A(0) =  2.5052394568638987E+00
A(1) =  1.2830816024602694E+00
A(2) =  2.2655669890401081E-01
A(3) =  1.3058266628015777E-01
A(4) = -2.0229823620712177E-02
A(5) =  3.9113765898126695E-03
```

$$\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}} \approx \frac{1}{a_0 + a_1x^2 + a_2x^4 + \cdots + a_5x^{10}},\ -\infty<x<\infty$$

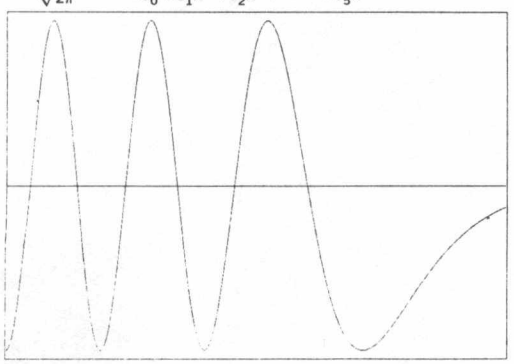


Figure 12

```
Minimax Fit
Function: (1/SQRT(2*PI))*EXP(-X*X/2)
  Range: 0.0 to 6.5
Maximum Absolute Error = 2.32244563E-09
Error Variance = 2.58681580E-18 Coefficients:
A(0) =  2.5066282606021772E+00
A(1) =  1.2533150158068533E+00
A(2) =  3.1332010186080966E-01
A(3) =  5.2251507235153920E-02
A(4) =  6.4745038039304478E-03
A(5) =  7.0651126222071438E-04
A(6) =  2.0618249722321060E-05
A(7) =  1.7799505117036926E-05
A(8) = -3.6392970226935947E-06
A(9) =  7.6125309644419998E-07
A(10) = -9.9471514152083657E-08
A(11) =  9.3097523153405433E-09
A(12) = -5.8145287978472377E-10
A(13) =  2.3753990217929335E-11
A(14) = -5.6523614575329404E-13
A(15) =  6.2370941540151992E-15
```

$$\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}} \approx \frac{1}{a_0 + a_1x^2 + a_2x^4 + \cdots + a_{15}x^{30}},\ -\infty<x<\infty$$

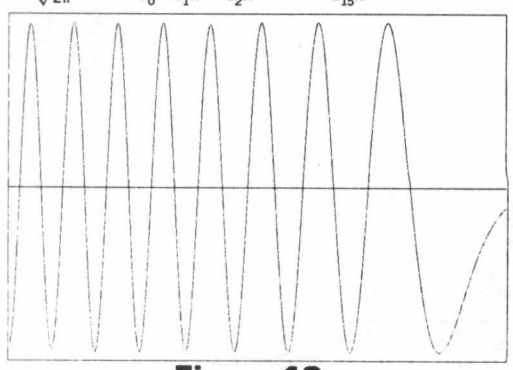


Figure 13

```
Minimax Fit
Function: (PI/2 - ARCSIN(X))/SQRT(1-X)
  Range: 0.0 to 1.0
Maximum Absolute Error = 2.18035876E-08
Error Variance = 2.36955457E-16
Coefficients:
A(0) =  1.5707963050017066E+00
A(1) = -2.1459880378710242E-01
A(2) =  8.8979047458192392E-02
A(3) = -5.0174698127643524E-02
A(4) =  3.0893003101670889E-02
A(5) = -1.7089738000951291E-02
A(6) =  6.6712412050773488E-03
A(7) = -1.2628162735097773E-03
```

$$\text{ARCSIN}(x) \approx \frac{\pi}{2} - \sqrt{1-x}\,(a_0 + a_1x + \cdots + a_7x^7),\ 0\leq x\leq 1$$

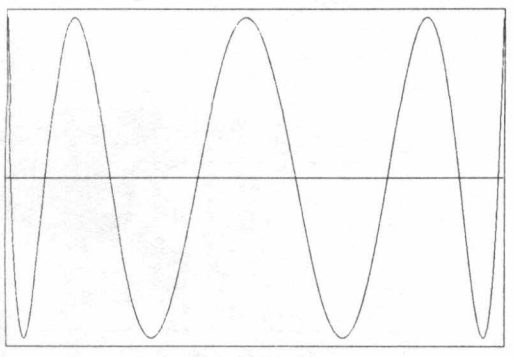


Figure 14

```
Minimax Fit
Function: (PI/2 - ARCSIN(X))/SQRT(1-X)
  Range: 0.0 to 1.0
Maximum Absolute Error = 1.60433862E-12
Error Variance = 1.27557003E-24
Coefficients:
A(0) =  1.5707963267933036E+00
A(1) = -2.1460183603317504E-01
A(2) =  8.9048588781845170E-02
A(3) = -5.0792024847596283E-02
A(4) =  3.3671623513562537E-02
A(5) = -2.4303150171847927E-02
A(6) =  1.8329760127140188E-02
A(7) = -1.3750906300991303E-02
A(8) =  9.5269440415934169E-03
A(9) = -5.5316843097328707E-03
A(10) =  2.4006330400566690E-03
A(11) = -6.6846664636149031E-04
A(12) =  8.7754386863194461E-05
```

$$\text{ARCSIN}(x) \approx \frac{\pi}{2} - \sqrt{1-x}\,(a_0 + a_1x + \cdots + a_{12}x^{12}),\ 0\leq x\leq 1$$

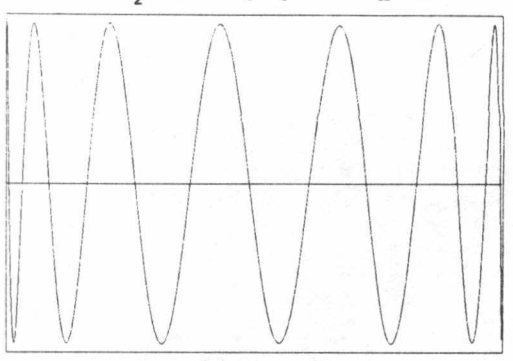


Figure 15

```
Minimax Fit
Function:  LN(1+X)      Range: 0.0 to 1.0
Maximum Absolute Error = 3.21271115E-08
Error Variance = 5.14899243E-16
Coefficients:
A(0) =  9.9999642149771728E-01
A(1) = -4.9987400637711582E-01
A(2) =  3.3179798745075179E-01
A(3) = -2.4072984401903777E-01
A(4) =  1.6764614981179890E-01
A(5) = -9.5320711256370566E-02
A(6) =  3.6083546727710289E-02
A(7) = -6.4523953561366849E-03
```

$$LN(1+x) \approx a_0x + a_1x^2 + \cdots + a_7x^8,\ 0 \leqslant x \leqslant 1$$

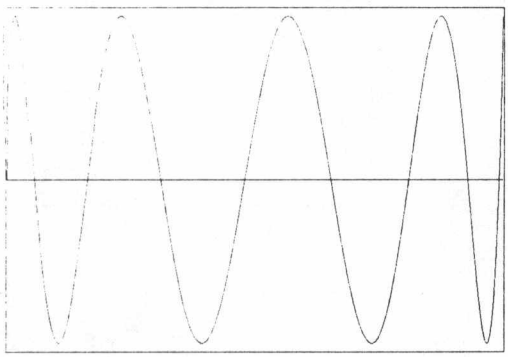


Figure 16

```
Minimax Fit
Function:  LN(1+X)      Range: 0.0 to 1.0
Maximum Absolute Error = 2.99005943E-12
Error Variance = 4.44361184E-24
Coefficients:
A(0) =  9.9999999921292859E-01
A(1) = -4.9999993436563822E-01
A(2) =  3.3333141255453208E-01
A(3) = -2.4997150728221187E-01
A(4) =  1.9974825550112713E-01
A(5) = -1.6521637838885567E-01
A(6) =  1.3708823785361379E-01
A(7) = -1.0851242464256000E-01
A(8) =  7.6176582881979454E-02
A(9) = -4.3423490764673826E-02
A(10) =  1.8141724720925251E-02
A(11) = -4.8159341987676569E-03
A(12) =  6.0063748040789543E-04
```

$$LN(1+x) \approx a_0x + a_1x^2 + \cdots a_{12}x^{13},\ 0 \leqslant x \leqslant 1$$

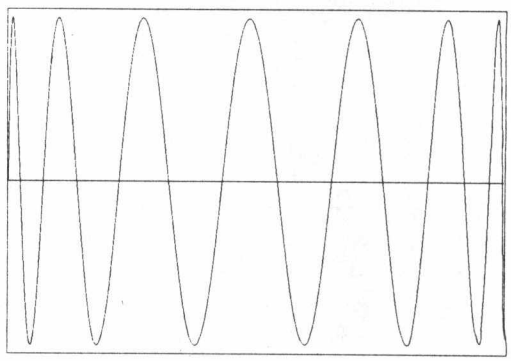


Figure 17

```
Minimax Fit
Function: EXP(-X) Range: 0.0 to 16.0
Maximum Absolute Error = 2.42142191E-07
Error Variance = 2.61153603E-14
Coefficients:
A(0) =  2.4999867539512141E-01
A(1) =  3.1257613051553959E-02
A(2) =  2.5913379887911865E-03
A(3) =  1.7157681582990019E-04
A(4) =  5.4274220448181193E-06
A(5) =  6.9077386115360822E-07
```

$$e^{-x} \approx \frac{1}{(1 + a_0x + a_1x^2 + \cdots + a_5x^6)^4},\ 0 \leqslant x < \infty$$

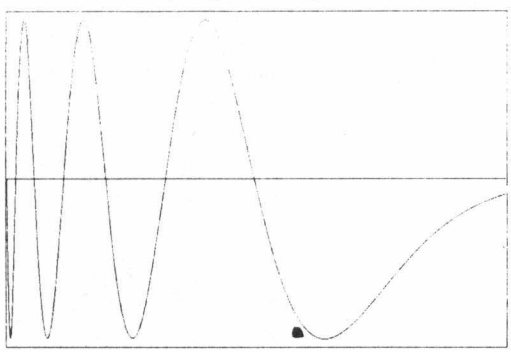


Figure 18

```
Minimax Fit
Function: EXP(-X) Range: 0.0 to 20.0
Maximum Absolute Error = 3.42649682E-13
Error Variance = 6.04906522E-26
Coefficients:
A(0) =  2.4999999999648442E-01
A(1) =  3.1250000041451426E-02
A(2) =  2.6041665079503278E-03
A(3) =  1.6276070164139176E-04
A(4) =  8.1377395547909460E-06
A(5) =  3.3925041987012053E-07
A(6) =  1.2048462387673388E-08
A(7) =  3.9315765909007019E-10
A(8) =  8.2512108449021085E-12
A(9) =  4.8158149678826166E-13
A(10) = -6.5346922729955708E-15
A(11) =  4.8211465914063989E-16
```

$$e^{-x} \approx \frac{1}{(1 + a_0x + a_1x^2 + \cdots + a_{11}x^{12})^4},\ 0 \leqslant x < \infty$$

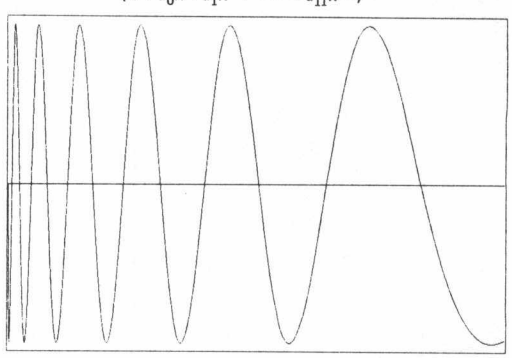


Figure 19

```
Minimax Fit
Function: 1-ERF(X) Range: 0.0 to 4.0
Maximum Absolute Error = 2.59522537E-07
Error Variance = 2.81576631E-14
Coefficients:
A(0) =  7.0523079991967080E-02
A(1) =  4.2281992401241905E-02
A(2) =  9.2705995340344790E-03
A(3) =  1.5190656505008346E-04
A(4) =  2.7663627971689751E-04
A(5) =  4.3048078931175112E-05
```

$$ERF(x) = 1 - \frac{1}{(1 + a_0x + a_1x^2 + \cdots + a_5x^6)^{16}},\ 0 \leqslant x \leqslant \infty$$

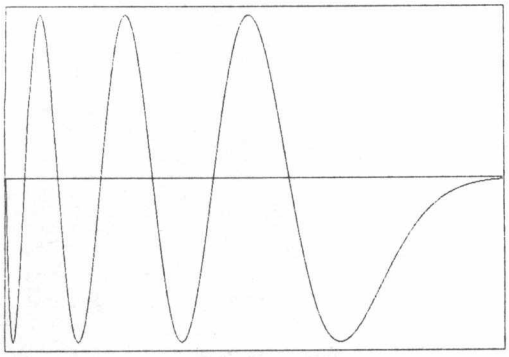


Figure 20

```
Minimax Fit
Function: 1-ERF(X) Range: 0.0 to 4.2
Maximum Absolute Error = 1.41490525E-12
Error Variance = 1.02491312E-24
Coefficients:
A(0) =  7.0523697931302184E-02
A(1) =  4.2275532274750102E-02
A(2) =  9.2877883593596861E-03
A(3) =  1.4906829666204733E-04
A(4) =  2.3210702049655493E-04
A(5) =  1.0664165183504473E-04
A(6) = -3.4002731629183212E-05
A(7) =  4.8035866639544990E-06
A(8) =  2.8758813985644038E-06
A(9) = -1.8271251310999870E-06
A(10) =  5.5554105470722536E-07
A(11) = -1.0046197131329913E-07
A(12) =  1.0438113302017026E-08
A(13) = -4.7894501157677281E-10
```

$$ERF(x) = 1 - \frac{1}{(1 + a_0x + a_1x^2 + \cdots + a_{13}x^{14})^{16}},\ 0 \leqslant x \leqslant \infty$$

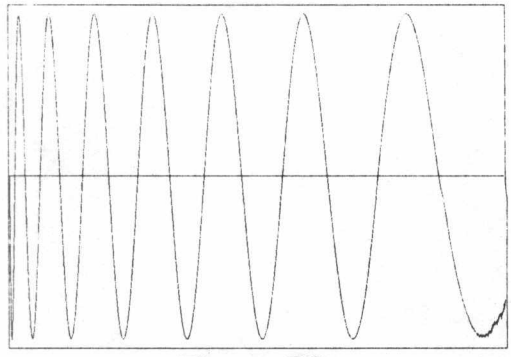


Figure 21

(Continued from page 91)

into the calculation. Making Qo large minimizes this error. There are, however, ways to completely avoid this error, which should make the SLS algorithm at least as accurate as algorithms such as Orthogonal Decomposition. But that is a topic for another article. ■■J

Minimax (Text begins on page 84)

Listing One

```
100 '   MINIMAX VIA ERROR FLUFFING ALGORITHM by Steven A. Ruzinsky
      This program demonstrates the determination of polynomial
      coefficients to a minimum maximum absolute error criterion. The result
110 ' is better than that of Hastings, Approximations for Digital Computers,
      1955, Princeton University Press, p 138.  This program is in IBM Basic.
120 ' ------------------------------------------------------------------------

130 DEFDBL A-Z : DEFINT I-N
140 N = 3 : M = 50 : ITERATIONS = 1000
150 DIM X(M,N) , Y(M) , XK(N) , Q(N,N) , QX(N) , AK(N) , A(N)
160 ' ------------------------------------------------------------------------
170 ' The following fills matrices Y(i) and X(i,j) with data.  The function used
      is SIN(PI*X/2), however, the data is modified so that the minimax
      criterion is applied to the relative error :
180 FOR I = 1 TO M
190 Y(I) = 1
200 E = I/M
210 D = 1/SIN(1.570796327#*E)
220 FOR J = 1 TO N
230 X(I,J) = D*E^(J+J-1)
240 NEXT J , I
250 ' ------------------------------------------------------------------------
260 ' The following initiates the Q matrix.  It may be necessary to
      to adjust the number in line 280 for best results.
270 FOR I = 1 TO N
280 Q(I,I) = 1000000!
290 NEXT I
300 ' ------------------------------------------------------------------------
310 ' The following loop with index, K, reiterates the sequential least squares
      algorithm.  Up to limit M, each data point is incorporated once into Q and
      AK. This results in a least squares fit to the data. Afterwards, the data
320 ' corresponding to the  maximum absolute error are reincorporated back into
      Q and AK.
330 EBEST = 1
340 FOR K = 1 TO ITERATIONS + M
350 IF K > M THEN GOSUB 740 ELSE GOSUB 650
360 D = 1
370 FOR J = 1 TO N
380 QX = 0
390 FOR I = 1 TO N
400 QX = QX + XK(I)*Q(J,I)
410 NEXT I
420 QX(J) = QX
430 D = D + XK(J)*QX
440 NEXT J
450 FOR J = 1 TO N
460 QX = QX(J)/D
470 FOR I = 1 TO N
480 Q(I,J) = Q(I,J) - QX(I)*QX
490 NEXT I , J
500 FOR J = 1 TO N
510 QX = 0
520 FOR I = 1 TO N
530 QX = QX + XK(I)*Q(J,I)
540 NEXT I
550 AK(J) = AK(J) + QX*E
560 NEXT J , K
570 ' ------------------------------------------------------------------------
580 ' The following prints the results :
590 PRINT "Coefficients:" : FOR I= 1 TO N
```

(Continued on next page)

Minimax (Listing Continued, text begins on page 84)

Listing Two

```
600 AK(I) = A(I)
610 PRINT AK(I)
620 NEXT I : GOSUB 740 : END
630 ' ---------------------------------------------------------------------------
640 ' Subroutine for incorporating each data point once :
650 E = Y(K)
660 FOR I = 1 TO N
670 XK = X(K,I)
680 XK(I) = XK
690 E = E - AK(I)*XK
700 NEXT I
710 RETURN
720 ' ---------------------------------------------------------------------------
730 ' Subroutine for finding maximum absolute error and corresponding data
    point :
740 EMAX = 0 : JMAX = 1
750 FOR J = 1 TO M
760 E = Y(J)
770 FOR I = 1 TO N
780 E = E - X(J,I)*AK(I)
790 NEXT I
800 E = ABS(E)
810 IF E > EMAX THEN EMAX = E : JMAX = J
820 NEXT J
830 PRINT "Iterations =", K-M, "Maximum Absolute Error =", EMAX
840 E = Y(JMAX)
850 FOR I = 1 TO N
860 XK = X(JMAX,I)
870 XK(I) = XK
880 E = E - AK(I)*XK
890 NEXT I
900 IF EMAX < EBEST THEN EBEST = EMAX : GOTO 940
910 RETURN
920 ' ---------------------------------------------------------------------------
930 ' Subroutine for saving best coefficients, A :
940 FOR I=1 TO N
950 A(I) = AK(I)
960 NEXT I
970 RETURN
```

End Listing One

Listing Two

```
201 LIST

 0 ( ------------ Minimax Algorithm, screen 1 of 7 ---------------)
 1 (      Minimax Via Error Fluffing by Steven A. Ruzinsky        )
 2 (  This program is written in Polyforth with FML Version 3.0   )
 3
 4 ( ------------ Matrix and Variable Definitions ----------------)
 5
 6 ( Enter range of x here : )       0.0 1.0
 7
 8 ( Enter dimension of X here : )   320 12
 9
10 2DUP matrices X x
11 LCONSTANT X2          LCONSTANT X1          0.0 LCONSTANT ek
12 CONSTANT n            CONSTANT m              0 CONSTANT k
13 0. 2CONSTANT i        1. 2CONSTANT 1.
14 m vectors y e         n vectors a A         n vectors xk CC
15 n vectors Qx xQ       xQ trnV             n n matrices Q QxxtQ
```

```
202 LIST

 0 ( ------------ Minimax Algorithm, screen 2 of 7 ---------------)
 1 : FUNCTION       ATAN ;
 2 : POLYNOMIAL     a CC **plyV ;
 3 : ERROR          FDUP FUNCTION FSWAP POLYNOMIAL F- ;
 4 : INITIALIZE     { X x y e A a xk Qx Q QxxtQ CC }clrA
 5                  1.0 ['] ek L! 0. ['] i 2! 0 ['] k ! ;
 6 : INSERT-DATA    X1 X2 y ['] FUNCTION yA
 7                  X1 X2 e xA
 8                  CC indA
 9                  CC f2* {A}
10                  1.0 CC f+ A.S
11                  e CC X f** .X. ;
12
13 : e-CALC        X a e *A+ y e f- A.A ;
14 : E-CALC        e abs max red{A} ;
15 : V-CALC        e sqr f+ red{A} m >N F/ ;
```

203 LIST

```
 0 ( ------------ Minimax Algorithm, screen 3 of 7 ---------------)
 1
 2 : ZERO       FDROP 0.0 ;
 3 : PLOT-ZERO  oldSCALE 0.0 1.0 @x ['] ZERO yA @x pltA newSCALE ;
 4 : PLOT-e     MRES dark
 5              BYline .95 SCALE e pltA PLOT-ZERO 1.0 SCALE ;
 6 : HPLOT-e    HRES dark BYline .95 SCALE X1 X2 ['] ERROR PLOT
 7              PLOT-ZERO FRAME 1.0 SCALE MRES BELL ;
 8 : BATCH-LEAST-SQUARES  X @ T@T   @ 1/A   X y a regA  ;
 9
10 : RESULTS    HPLOT-e 27 EMIT 69 EMIT
11 CR 0= IF ." Least Squares Fit" ELSE ." Minimax Fit" THEN
12 CR ." Function: ARCTAN(X)     " ." Range: 0.0 to 1.0"
13 CR ." Maximum Absolute Error =" 2TOP abs max red{A} 8 S.
14 CR ." Error Variance =" 2TOP sqr f+ red{A} 2TOP elm >N F/ 8 S.
15 CR ." Coefficients:" 16 SCI a  V. 16 FIX CR ;
```

204 LIST

```
 0 ( ------------ Minimax Algorithm, screen 4 of 7 ---------------)
 1
 2 : NORMALIZE-@    m >N  n >N 10.0 F@  F/  @ f@  A.S ;
 3
 4 : REDUCE-e  e-CALC
 5             a @x A>A V-CALC ['] ek L! 50 0 DO
 6             e-CALC
 7             X e A rregA
 8             A a f+ A.A
 9             V-CALC CR FDUP I . 16 S. ek 16 S.
10             FDUP ek F< IF ['] ek L! a @x A>A  ELSE FDROP THEN
11             LOOP
12             1.0 ['] ek L!
13             @x a A>A  ;
14
15 : x>X       m 0 DO I x I X R>R LOOP ;
```

205 LIST

```
 0 ( ------------ Minimax Algorithm, screen 5 of 7 ---------------)
 1
 2 : SPEEDUP      I' = IF m 2/ ['] m !   X A e @A+   y e f- A.A
 3                e abs {A}   e x V>C   x X srtsC   e y srtsA
 4                y revC   X revC   X x A>A   m n X !dim
 5                m 1 y !dim   m 1 e !dim   x>X  m n x !dim THEN ;
 6
 7 : FIND-MAX(e),xk;RECORD-BEST-a,ek
 8                                  e-CALC
 9                                  e amax{i}
10                                  ['] k ! FDUP
11                                  ek F< IF
12                                  FDUP ['] ek L!
13                                  a A A>A
14                                  DROP 0 THEN
15                                  k X 0 xk R>C ;
```

206 LIST

```
 0 ( ------------ Minimax Algorithm, screen 6 of 7 ---------------)
 1
 2
 3 : SEQUENTIAL-LEAST-SQUARES
 4                           xk @ x@ @T+
 5                           @ xk @x @A+
 6                           @x x@ @xx@ f@ .X.
 7                           xk @x @V+
 8                           1.0 F+ 1/N @xx@ f@ A.S
 9                           @xx@ @ -f A.A
10                           @ xk @x @A+
11                           k e V@
12                           @x f@ A.S
13                           @x a f+ A.A  ;
14
15
```

207 LIST

```
 0 ( ------------ Minimax Algorithm, screen 7 of 7 ---------------)
 1 : MINIMAX
 2             INITIALIZE
 3             INSERT-DATA
 4             BATCH-LEAST-SQUARES
 5             REDUCE-E  PAGE 0 RESULTS  BELL KEY DROP
 6             NORMALIZE-@
 7             0 50 0 DO 1000 0 DO  i  1. D+ ['] i 2!
 8             FIND-MAX(e),xk;RECORD-BEST-a,ek
 9             CR i 7 D.R  k 10 U.R DUP 7 U.R 5 SPACES 8 S.
10             SEQUENTIAL-LEAST-SQUARES
11             1+ LOOP
12             0 SPEEDUP 5 SPEEDUP
13             DARK PLOT-e BELL BELL BELL LOOP
14             A a A>A
15             PAGE 1 RESULTS ;
```

End Listings

For those interested in FML, version 3.0 is available at a cost of $150 ppd exclusively from United Applied Research, Inc., P.O. Box 1164, North Riverside, IL 60546, and requires the IBM PC with an 8087 and at least 128K bytes of RAM (320K bytes preferred) and Polyforth with 8087 support.

Languages and Parentheses

A Suggestion for Forth-like Languages

by Ernest E. Bergman

by Ernest E. Bergmann Physics, Building 16, Lehigh University, Bethlehem, PA 18015.

Why do most languages use parentheses, brackets, or braces? In this article, we shall consider this question (and even discuss a language that doesn't use them!) by tracing an informal route from the philosophy of language and communication to the practicalities of compiler construction.

This article will also serve as something of a followup to the brief description of the public domain Forth-like language PISTOL which appeared in *DDJ* in February 1983. PISTOL will be used as an example in the discussion; both its virtue and its vice is a lack of parentheses. The discussion should also be applicable to some degree, however, to most languages which use a Reverse Polish syntax.

Languages

Languages are for communicating. We use English to talk to each other. Fortran, developed as an aid to give computers instructions on how to carry out numerical calculations, was an early attempt at designing a language. Only a moderate success, Fortran's lack of regularity in syntax makes the design of the compiler's parser quite arduous; at the same time the programmer finds it difficult to remember the details of the syntax. BASIC then was developed, its syntax making it easier to learn and easier to parse.

A more recently developed language, Pascal has earned high marks for readability by both people and machines. The regularity of its syntax makes compiling easier and its readability by humans allows easier modification of the program ("maintenance"); that is, if you can understand the program, you are more likely to find its bugs and fix them, but if a program is incomprehensible, you will feel some inhibitions about expanding upon it.

Readability also means that a language can be a vehicle for interpersonal communications. (New algorithms are frequently described in Pascal.) Expressiveness is another important aspect of a language. It is easier to think of approaches to problems if the language supports useful concepts.

A "good" language, therefore, should be easy to write and capable of expressing the concepts needed for the implementation of algorithms. It should be easily read by people as well as by machines. We may not have a good language yet, but the present languages may yield fruitful approaches for better languages in the future.

Parentheses in Algebraic Languages

For numerical calculations, we often need to provide a formula, such as:

```
VOLUME=LENGTH*WIDTH*
       HEIGHT
```

For more complicated formulas expressed in this algebraic format, we must resort to using parentheses:

```
TOTAL=(L1*W1*H1)+(L2*W2*H2)
```

Since many algebraic languages, such as Fortran, BASIC, and Pascal, assign a higher precedence to multiplication than to addition, the parentheses here are optional as far as the computer is concerned. But for people to read this, the parentheses are helpful (even essential) for comprehension. If the formula requires that the calculation be carried out in an order different from the normal precedence, the parentheses become mandatory. Additional parentheses, in addition to the mandatory pairs, are acceptable and may improve readability.

Because of the increased readability for humans, parentheses are frequently used in the design of computer languages. To write in a language that uses parentheses requires considerable training. For example, people accustomed to a Hewlett-Packard calculator, which uses RPN (Reverse Polish Notation), would run into problems using an algebraic calculator, such as those produced by Texas Instruments. While many people write in parentheses pairs after the rest of the equation format is written out, many users of HP calculators would find the RPN hard to understand if it were written out.

In summary, algebraic notation is easier to read and harder to write, while RPN, which does not use parentheses, is easier to write (if it is written sequentially) but harder to read (a somewhat random-access process). To parse an expression serially requires more work for a compiler if algebraic notation is used than if RPN is used.

LISP

The use of parentheses in LISP (sometimes taken to stand for Lots of Irritating Single Parentheses) is very constrained. In list notation, parentheses are required but you may not add extra pairs optionally without altering the meaning. For example, the list A, B, C would be written (A B C), whereas the meaning of ((A B C)) is "the list containing the single element (A B C)"; it is a list of one list.

It is an interesting question whether an RPN version of LISP would be possible through constructions such as A B C LIST and A B C LIST LIST for (ABC) and ((ABC)), respectively. In addition to the question of ambiguities that might exist, we would have to assess the readability of the notation.

PISTOL

PISTOL was developed to make a Forth-like language available to users

on large mainframe computers. Because it has been written both in Pascal and in C, it should be available with little conversion effort on many other computers.

The development of PISTOL involved certain conscious departures from Forth, including the adoption of some of the methods used in a similar language, STOIC (Stack-Oriented Incremental Compiler), which was developed as an extension to Forth. It is a shame that STOIC has not received as much attention as Forth since it is well conceived and documented. The most noticeable difference between Forth and STOIC or PISTOL is the latter's use of string literals to regularize the syntax further toward RPN and to increase the ease of handling strings.

In any case, all three languages (Forth, STOIC, and PISTOL) are largely precedence free and use an RPN style for expressing computations. Any reader who uses an HP calculator will find the description of these languages, which do not rely on paired parentheses to control precedence, quite familiar, although programming an HP calculator usually is not done in writing. Suppose we wish to evaluate:

$$(5+7)*13-7$$

We would type (for STOIC, PISTOL, and Forth):

```
5 7 + 1 3 * 7 - =
```

No parentheses are used. You would read this as: "take 5 and 7; add them; take 13; multiply it by the previous result; take 7; subtract it from the previous result; display the answer."

RPN notation is easier to write because no "special cases" occur with regard to precedence. If you have a "function" that takes three arguments and returns two results, it "fits" into the notation without any adjustments (and still without the need for parentheses). Variable numbers of arguments and results can be handled as well; in contrast, the Pascal function WRITELN takes a variable number of arguments and must, consequently, be a built-in feature of the compiler.

Parsing and Code Generation

These days parsing methods readily applicable to algebraic expressions are well known. More often than not an intermediate code is produced, such as P-code. This code frequently is based upon some virtual stack machine.

Compiling expressions in a Forth-like language is extremely simple; you create code that pushes each literal onto a parameter stack as it is encountered in parsing the source code. Almost all other objects encountered in the parsing process simply require code that invokes the named function or procedure. The simplicity of the language reduces the programmer's work in writing source code: you do not have to remember the precedence of various operators since you are writing "directly" to the stack machine.

Program Flow in Structured Languages

Almost all software must exercise a conditional transfer of program control, in addition to purely sequential execution. For example, in the C language the general two-way conditional branch is:

```
if (expression)
{true actions}
else
{false actions}
```

If the original expression, enclosed in parentheses, evaluates to "true," only the true actions, enclosed in the first set of braces, are carried out. Otherwise, only the false actions, enclosed in the second pair of braces, are used. The "else" portion is optional. If only a simple statement is used for the true or false action, the enclosing braces become optional.

The analogous structure in Pascal, which is also subject to analogous simplifications, is:

```
IF boolean expression
  THEN BEGIN
    true actions
    END
  ELSE BEGIN
    false actions
    END
```

The key words, BEGIN and END, serve the same function as the braces do in C. The boolean expression need not be enclosed in parentheses. The same need to group actions together exists for the Pascal constructs WHILE, REPEAT, FOR, PROCEDURE, and FUNCTION.

A common feature of most languages is the use of keywords and/or some sort of brackets or braces to delimit the various elements that form the pieces of these control structures.

PISTOL *Structures*

PISTOL is no exception. It groups the elements or actions by using pairs (and occasionally triplets) of keywords. The following elements are defined as shown:

1. Procedure or function:
   'name : body of definition ;
2. Two-way branch (the "else" clause optional):
   condition IF true actions ELSE false actions THEN
3. FOR loop:
   upper-lim initial-val DO body LOOP
4. REPEAT loop:
   BEGIN actions condition END
5. WHILE loop:
   BEGIN condition IF actions REPEAT

The "case" construction (actually closer to the COND of LISP) is for the benefit of humans:

```
variable OFCASE
        condition C: actions; C
        condition C: actions; C
        .
        .
        .
        ENDCASE
```

PISTOL's *Syntax Checking*

During compilation, the PISTOL compiler is particularly careful to verify that the user is following the few syntactical rules that do exist. To test for proper nesting, the compilation process uses a check stack. When a word that starts a structure is encountered, a character of the appropriate type is placed on this stack, and when a word that completes a structure is encountered, a character is removed from the stack and checked to make sure it is of the right type.

Consider the following fragment of

PISTOL code:

```
... : ... DO ... BEGIN ... END ...
        LOOP ... ;
```

In this example, the : leaves a ":" on the check stack; the DO leaves a "D" on the stack; and the BEGIN leaves a "B" on the stack. When the END is encountered, the check stack is popped (returning the "B" from BEGIN) to see whether a BEGIN was used last. Later the LOOP is encountered, and the stack is popped and checked for "D." Lastly, the ; is encountered, so the popped character is checked to be a ": ."

Not only does the check stack ensure proper nesting has been maintained (an incorrect nesting is likely to produce code that would catastrophically crash), but it also enables the compiler-interpreter to ascertain when an input line completes the outermost structure; it may then proceed to interpretation (execution) of the code.

As an aid to the user, PISTOL prompts with a display of the current contents of the check stack. The user sees exactly into which structures he or she has entered and still must complete.

Brackets Proposal

I propose that a new structure be available to Forth-like languages: braces (curly brackets). They would be optional but could be inserted for improved clarity. Let's examine an example taken from one of the basic definitions in PISTOL (from the file PBASE2):

```
'INDENT : DUP TERMINAL-
          WIDTH W@ LT IF
          COLUMN W@ -
          SPACES
          ELSE IFCR DROP
          THEN ;
```

With the addition of braces; we would have:

```
'INDENT : DUP { TERMINAL-
          WIDTH W@ LT } IF
          { COLUMN W@ -
          SPACES } ELSE
          { IFCR DROP } THEN
          ;
```

Here the readability is improved in that what IF is testing is delineated as well as the two alternative conditional courses of action. Forth-like languages are sometimes criticized because everything runs together; could these braces form the solution?

Another benefit of braces is the deferral of both execution and the recycling of string space. For example:

```
X> "HELLO THERE" MSG
HELLO THERE
```

properly prints the greeting, whereas:

```
X> "HELLO THERE"
1X> MSG
MSG
```

obviously does not! In the second example, the "HELLO THERE" pushes a pointer to itself on the parameter stack. (We see from the second prompt that there is now one item on the stack.) However, the pointer points to an area where the space has been recycled, so the (erroneous) string "MSG" is printed instead. With braces this recycling action can be delayed, as follows:

```
X> {
X{> "HELLO THERE"
X{> MSG
X{> }
HELLO THERE
```

The prompt indicates that we have entered the "brace structure." Thus we need not worry about ending lines after we have typed in (possibly quite long) strings needed for execution on the following lines; we simply must remember to encircle the strings and the words that use them with braces.

I have included a listing of a file BRACKET that adds the needed definitions to this structure (Listing One, page 104). Note how little work is needed to extend Forth-like languages! I also have included the transcript of a session at the terminal that shows what happens when BRACKET is loaded, as well as a number of facilities that are part of PISTOL (Listing Two, page 106).

Conclusion

Forth-like languages such as PISTOL demonstrate that delimiters such as parentheses or braces are not required for a general-purpose computer language. (Such languages, however, may use them for other things. Forth implementations, for example, often use parentheses and square brackets for special purposes.) With the recent emphasis on readability by people even at the expense of greater effort in writing, perhaps braces should be available for optional use by the writer. The language compiler could ignore these braces except perhaps to check the syntactical requirements that structures properly "nest," as with PISTOL here. Whether or not such a feature can be added to a particular Forth system as easily as it has been added here to PISTOL probably depends on details of the particular implementation.

PISTOL is available on a CP/M diskette at nominal cost from the C User's Group, Box 287, Yates Center, KS 66783, and as SIG/M Volume 126 from the Amateur Computer Group of New Jersey (ACGNJ), Box 319, South Bound Brook, NJ 08880.

■■J

Languages and Parentheses (Text begins on page 102)

Listing One

```
% OCTOBER 2, 1983 ****BRACKET DEFS FOR PISTOL*****
% THE TOKENS, { and } ARE NOPS IN USE BUT ENABLE
% THE PROGRAMMER TO SUPPLY EMPHASIS TO GROUPINGS

HEX

% IMMEDIATE PATCHES DEFINITION TO EXECUTE DURING COMPILATION:
'IMMEDIATE : 'NOP ADDRESS CURRENT W@ W@ W- W! ;

'SYNTERR : "SYNTAX ERROR" MERR ;

'{ : SYNTAXBASE C@ 1+ DUP CHKLMT LT
     IF SYNTAXBASE C!                  % UPDATE COUNT
        7B SYNTAXBASE DUP C@ + C!      % "PUSH" '{'
     ELSE SYNTERR
     THEN
   ; IMMEDIATE

'} : SYNTAXBASE DUP C@ + C@ 7B EQ
     IF SYNTAXBASE C@ 1- SYNTAXBASE C!
     ELSE SYNTERR
     THEN
   ; IMMEDIATE

;F
```

Listing of BRACKET, which adds the definitions of "{" and "}" to PISTOL.

End Listing One

Listing Two

The enclosed terminal session listing is provided to demonstrate a number of important features and aspects of PISTOL.

Here is a running commentary on that listing. The only hint that this was run on a CP/M environment appears on the first line with the system prompt A>. (All subsequent dialogue is standard for all PISTOL environments.) Restoring the "core image," CORE2, is a fast way to recreate PISTOL without recompiling the extensive source code that is supplied originally to bring PISTOL up. Turning the ECHO ON enables us to watch the lines of the file BRACKET being input for compilation.

BRACKET is being loaded; the following lines should look very much like the supplied listing of BRACKET, but here PISTOL is supplying the prompts. The word % is used to mark the rest of the line as a comment. Notice that after a couple of lines the prompt changes from X> to H> to show that the number base has been switched from decimal to hexadecimal (by the word HEX).

The word IMMEDIATE is defined to poke the address of NOP into a location that has to be calculated with respect to the start of the code of the CURRENT definition. The operator W@ does a word fetch; the W- backs up an address by one word size; W! pokes the word (here, the address of NOP).

After SYNTERR is defined we see { being defined. Since the definition takes several lines, we see that the prompt contains a : , indicating that the : ... ; structure is incomplete; also, within the same definition, an IF...ELSE...THEN structure is used and the "F" and "E" tokens appear in the prompt. The number that starts the prompt indicates that the parameter stack is not empty; it is used in the compiling process to calculate the address offsets needed to generate the code associated with most of the structures.

The word ;F is the logical end-of-file; PISTOL confirms that the loading of the file is complete.

I have typed TOP10 to get a list of the 10 most recent definitions. Since I am not interested in seeing the next 10 most recent definitions, I throw away (DROP) the backward link left on the stack. DECIMAL returns me to base 10.

It is possible to disassemble a definition with the built-in disassembler DIS; IMMEDIATE is analyzed to look more

or less like its original definition.

Finally, I do a few tests of the definitions of { and of }. It does appear that the prompt is properly indicating the number of brackets, and when I type too many }'s, an error message is displayed. Also checked here is the way strings are recycled and that the { can be used to defer execution until the matching } is typed.

Lastly, the TRACE facility is demonstrated for the word MSG. BYE is the proper way to exit PISTOL.

```
A>PISTOL
*** PISTOL 2.0 ***
X>'CORE2 RESTORE
X> ECHO ON
X> 'BRACKET LOAD
X> % OCTOBER 2, 1983 ****BRACKET DEFS FOR PISTOL*****
X> % THE TOKENS, { and } ARE NOPS IN USE BUT ENABLE
X> % THE PROGRAMMER TO SUPPLY EMPHASIS TO GROUPINGS
X>
X> HEX
H>
H> % IMMEDIATE PATCHES DEFINITION TO EXECUTE DURING COMPILATION;
H> 'IMMEDIATE : 'NOP ADDRESS CURRENT W@ W@ W- W! ;
H>
H> 'SYNTERR : "SYNTAX ERROR" MERR ;
H>
H> '{ : SYNTAXBASE C@ 1+ DUP CHKLMT LT
1H:>       IF SYNTAXBASE C!                  % UPDATE COUNT
2H:F>          7B SYNTAXBASE DUP C@ + C!    % "PUSH" '{'
2H:F>        ELSE SYNTERR
2H:E>        THEN
1H:>     ; IMMEDIATE
H>
H> '} : SYNTAXBASE DUP C@ + C@ 7B EQ
1H:>       IF SYNTAXBASE C@ 1- SYNTAXBASE C!
2H:F>        ELSE SYNTERR
2H:E>        THEN
1H:>     ; IMMEDIATE
H>
H> ;F
BRACKET LOADED
H>
H> TOP10
687C    }
684A    {
683A    SYNTERR
6820    IMMEDIATE
67EE    HELP
66DA    TYIL
668A    UC
667C    Q'
666E    A'
65EC    FINISH
1H> DROP
H> DECIMAL
X> 'IMMEDIATE DIS
'IMMEDIATE [:]  "NOP"    ADDRESS CURRENT W@       W@       W-
W!       ;
X>
X> {

X{> { { {

X{{{{> } }
```

Languages and Parentheses (Listing Continued, text begins on page 102)

Listing Two

```
X{{> }

X{> }
X> }
SYNTAX ERROR
*** PISTOL 2.0 ***
X> "HELLO THERE" MSG
HELLO THERE
X> "HELLO THERE"
1X> MSG
MSG
X> { "HELLO THERE"

X{> MSG

X{> }
HELLO THERE
X> 'HELLO 'MSG TRACE
'MSG BEING TRACED:
(1)   32086                                    DUP
(2)   32086   32086                            C@
(2)   32086   5                                LINE-SPACE?
(1)   32086                                    DUP
(2)   32086   32086                            1+
(2)   32086   32087                            SWAP
(2)   32087   32086                            C@
(2)   32087   5                                TYPE  HELLO
(0)                                            (;)
TRACE COMPLETED
X> BYE

PISTOL NORMAL EXIT
```

End Listings

Balancing Act:

The Ultimate Checkbook Balancing Program

Balancing a checkbook is a little like asking which is the best programming language: every time you do it, you're liable to get a different answer. But now, after long years of expensive research, that problem is behind us. This article won't just describe what may be the world's easiest-to-use checkbook balancing program—the revolutionary program itself is included here free. (See the listing below.)

Everyone seems to have trouble with checkbooks and banks. Do you find yourself struggling with Stith's first law of banking: No Deposit, No Return? Do you wonder how banks can make you stand in line and then demand a service charge? How often are you told "The check is in the mail"? Have you ever received a dirty look from a bank guard when you've joined a long line and asked, "Hey, what's the holdup?"? If so, a stupid program isn't going to solve anything. But if you just need an infinitely simpler way to balance your checkbook, read on.

by John E. Stith

John E. Stith, P.O. Box 6677, Colorado Springs, CO 80934.

This program has all the traditional boring features. It runs rapidly, uses minimal storage space, has adequate comments, and is easy to convert to other BASICs. (It's written in Microsoft BASIC.) But I didn't stop there. No.

Not only does it work with paper *and* rubber checks, you never again have to record a checking transaction. In fact, you don't even need an active account. How many programs require rerunning every single month? Not this one. Run it once and you're done—it does it right the very first time.

It also runs perfectly with personalized checkbooks, as long as your name is printed in lightweight letters. Cowhide, pigskin, hamster hide—no problem. It works with both rectangular and square checkbooks. An ambitious clever programmer could probably even modify it to handle *round* checkbooks. Finally, the new, improved version 17.0 adapts automatically to whatever units you prefer. Enter your dimensions in inches, microns, light years, and the program will deliver your answer in those very same units.

So, say goodbye to early withdrawal symptoms and automated tellers that don't smile back at you. Here's your chance to let modern technology solve more problems than it creates.

■■J

Balancing Act Listing

```
100 REM BALANCING ACT -- THE ULTIMATE CHECKBOOK BALANCING PROGRAM
110 REM AUTHOR -- JOHN E. STITH -- c COPYRIGHT 1983
200 REM L = CHECKBOOK LENGTH
210 REM W = CHECKBOOK WIDTH
300 PRINT "BALANCING ACT -- THE ULTIMATE CHECKBOOK BALANCING PROGRAM"
310 PRINT
400 PRINT "WHAT IS THE LENGTH OF YOUR CHECKBOOK (THE LONG DIMENSION)";
410 INPUT L
500 PRINT "WHAT IS THE WIDTH OF YOUR CHECKBOOK (THE NARROW DIMENSION)";
510 INPUT W
550 PRINT
600 PRINT "NOW PLACE YOUR CHECKBOOK SO ITS LONG DIMENSION FACES YOU."
610 PRINT "MARK THE SPOT"; L/2 ; "UNITS FROM THE LEFT SIDE ";
620 PRINT "AND"; W/2 ; "UNITS FROM THE TOP."
630 PRINT "THIS SPOT IS AS CLOSE AS YOU ARE GOING TO GET TO ";
640 PRINT "THE CENTER OF GRAVITY."
650 PRINT "PLACE YOUR PENCIL POINT UNDER IT, ";
660 PRINT "AND YOUR CHECKBOOK SHOULD BALANCE."
670 PRINT "DON'T WORRY IF IT DOESN'T BALANCE PERFECTLY.  ";
680 PRINT "EVERYONE HAS THAT PROBLEM."
999 END
```

End Listing

C Programmer's Library

by Jack Purdum, Timothy Leslie, Alan Stegemoller
Published by Que Corporation
$19.95, 366 pages
Reviewed by John R. Johnson

The definitive source on the C programming language is, and will likely remain, the book *The C Programming Language* by Brian Kernighan and Dennis Ritchie. It is elegant, complete, and spare. Needless to say there has been a rush to publish expansions, explanations, and elucidations.

One of the best of these was the *C Programmers Guide* by Jack Purdum. This second book also issues from the senior staff at Ecosoft Incorporated, and seems to continue where the first left off.

The declared audience for the book is a programmer who is familiar with the syntax of the C language and wishes to learn how to develop the full power of C. Rather as a bonus, they throw in the sample source code they use to demonstrate proper use of declarations, structures, and functions for library construction. This is code for an ISAM data base system with a significant application example, a book catalog and retrieval system.

It is apparent that several authors were involved. The initial coverage of complex declarations was thorough and well done. I thought it a good idea to tie the declaration followthrough to an actual compiler algorithm. A good understanding of the underlying mechanism makes variations in syntax more understandable.

The coverage of structures, on the other hand, was much more subtle. Rather than just saying "here is a structure" the author let you become aware of what using structures correctly does for your programs. This directed self-discovery is also a powerful tool for communication of understanding. The chapter on sorting is good, not for the sorting information, but for the separation of functions. The reader is shown how to improve average programs by taking advantage of the features of the C language.

Although all of the authors are associated with the same company, which does produce a C compiler system, you would not know that from the treatment. They have an appendix with samples of the library functions that have to be added to compile the sample programs with most of the popular C compilers available for micros. As an added bonus most of the sample programs were compiled and tested with the UNIX PCC compiler.

Overall, I found the book a useful adjunct to my Kernighan and Ritchie. I think that most serious users of C would find it worth having the samples in a program library and the book on the shelf.

Building Controls Into Structured Systems

by Alan E. Brill
Published by Yourdon Press
Reviewed by Dr. Joseph B. Rothstein

As microcomputer programming has moved from the basements of hackers to the boardrooms of the Fortune 500, the demand for accountability and control for microcomputer programs has grown almost as fast as the microcomputer industry itself.

The thought that programs run on micros would be important enough to warrant auditing would have drawn laughter just a few years ago. But as successive generations of ever-more-powerful microcomputers have become an essential part of corporate life, comptrollers and financial managers have come to insist on the same sort of tight controls and auditing procedures they are used to on their mainframe and minicomputer systems.

The systems analysts, auditors, and financial management specialists at Yourdon, Inc. have established themselves among the leading consultants to big business. They embraced the concepts of structured programming early on, and have built a reputation on their seminars, periodicals, and book titles.

A newly published text from Yourdon Press, *Building Controls Into Structured Systems* by Alan E. Brill condenses some of the most important concepts of EDP auditing, applications systems controls, and essential procedures into fewer than 150 pages. While not aimed specifically at micro users or programmers (specific hardware is not even mentioned), the concepts involved are as appropriate to a "Little Wonder-80" as they are to an IBM 4300.

This is not an accounting book; nor is it a book about programming — indeed, there is not a single line of program code in the entire volume. Rather, Brill describes a methodology for building well-controlled systems featuring internal accounting controls and external application controls, all designed to provide reasonable assurance that the system is able to detect and handle any errors or anomalies that may crop up.

Numerous examples, including fictional "dialogues" between programmers and users, auditors, or managers, illustrate the importance of such controls. As more and more businesses automate their payrolls, customer lists, etc., the risks of uncontrolled systems are all too often illustrated by fiscal tragedy. Such horror stories seldom make headlines; most businesses would rather take their lumps and swallow their losses than have the world learn how lax they've been. But most programmers in large organizations can describe in gory detail how some other guy (never themselves!) resigned in disgrace after his or her system fell

apart. It's just this sort of tragedy that the author counsels programmers how to avoid.

As the title implies, structured techniques are emphasized throughout. *Just as these* techniques can be applied to the task of designing and implementing program code, so can they be applied to auditing and application controls. In the first few chapters Brill introduces the field of EDP auditing and the concepts of phase-related control — the process of identifying, specifying, and documenting the internal controls appropriate to each stage of the system's development life cycle. Successive chapters cover modeling controls during the analysis, design, and implementation phases, followed by chapters on documentation reviews and maintenance reviews. The book closes with a discussion of several typical plans for action that are almost sure to fail, then contrasts those with an approach that is more likely to succeed.

A breezy conversational style is used throughout, enlivening a subject that could easily have been treated in an academic and jargonistic fashion. Drawings, charts, and summaries are used extensively to further clarify important concepts.

Clearly, though, this is not a book for everyone. The hobbyist, game designer, or personal-software user may never need or want such stringent controls. But once money enters the picture, especially big money — and particularly *corporate* big money — the controls and audit procedures Brill describes are likely to become *de rigueur.* The serious professional programmer had best acquaint him or herself with them.

As the author states in his Preface, "You *will* eventually have to put controls into your application systems. You can choose to wait until the auditors and your boss force you to do so, but wouldn't it be easier and more impressive if you took the initiative to *assure* that effective internal controls are a built-in part of every application system that you build?" It's just this sort of no-nonsense, practical approach that makes *Building Controls Into Structured Systems* a worthwhile introduction to a subject that is moving to the forefront among the concerns of business executives and EDP planners, and thus will influence the careers and futures of programmers as well.

MENTOR — The Magazine on Disk

Mentor Computer Series, 533 Sutter Street, Suite 914, San Francisco, CA 94102
$18.00 per issue
Reviewed by Dian Crayne

The idea of putting magazines on disk is one of those notions that have been kicking around, under one guise or another, ever since computers came out of the office and into the home.

Most publishing companies have shied away from the diskette format because of the problems involved. Not only would they have to go through all of the monumental work involved in putting out a magazine, they would also have to make sure that it got transferred (without errors) to a flexible, rather fragile diskette and sent to a subscription list without being folded, spindled, mutilated, or erased. The idea alone is enough to make most publishers turn pale.

One company that has faced up to the challenge is Mentor Computer Services, which published the fourth issue of *Mentor,* "the magazine on disk," in January of this year. Although some other publishers — notably Ziff-Davis with its *PC Disk* — have flirted with the concept of sending out software on magnetic media, *Mentor* appears to be the only company that has really gotten off of the ground so far.

"Disk is more appropriate media than paper for computers," says editor Ted Lester, whose staff does both editing and publishing for this venture. He went on to explain that he is particularly proud of the fact that *Mentor* has published some excellent add-on utilities for existing programs, which gives subscribers software they can actually use instead of curiosities to stuff away on a shelf.

The first issue of *Mentor,* for instance, carried a WordStar customizing program; follow-up articles appeared in the next two issues. Issue No. 4 also carried LEADS, "The Buyer's Worksheet for Lotus 1-2-3," a worksheet program that helps salesmen to track customers and sources. *Mentor* has also published mailing list programs and a MailMerge enhancement package.

Other articles and programs that appeared in issue No. 4 were "Critical Path Method Project Schedule," by Philip Jacka; "Tally," a data base counting program by Richard Malm; a discussion on "Transfers Between dBase II and 1-2-3" by LeBlond and Cobb; and "Clock," an assembly language tutorial program that shows users how to display the current time on their screen.

There was also an article on PCrayon, a discussion of job streaming for Pascal compilers, and some regular features including a column by Rick Albert, letters to the editor, editorials, and a shoppers' reference list to products mentioned in the magazine. And yes, there is a cumulative index.

Mentor comes in a sturdy black plastic diskette box, the kind that opens up to make a vertical file for the diskettes. New subscribers get a startup diskette and one or more program diskettes, which hold the contents of the magazine. Since DOS is not distributed with the magazine (diskzine?), users have to create their own system diskette and then move the required files to it from the startup diskette. Once you've gotten the starter set, subsequent issues consist only of the magazine diskettes themselves.

Mentor looks a lot like a hard-copy magazine once you begin to display it on the screen. The first page displays the magazine logo, and the next few pages list the editorial staff and then the contents. Pages are turned by pressing the return (enter) key, and you can either page through the entire magazine or read the table of contents and select an article by its letter. There are even some advertisements, but unlike a lot of regular computer magazines on the stands, *Mentor*'s articles aren't submerged in an ocean of commercial ballyhoo.

Once you've selected an article to read, you can page through it, abandon it and select another one, or — if the

article is written in conjunction with a piece of software — actually run a program and get an immediate demonstration of the way it works. (The editorial column in issue No. 4 lists the code for a short animation routine, executes the routine, and then returns to a discussion of what happened.) This is the sort of immediate reinforcement that educators have been pursuing for years.

Editor Lester regards this ability to run programs within the boundaries of *Mentor* as one of the most exciting facets of the magazine. He has plans to take the magazine's educational aspects even further by establishing a subscriber's remote bulletin board system, where programming techniques and new advances in electronics can be discussed in open forum.

Another advantage *Mentor* has over traditional computer magazines is the storage space it saves: *Mentor*'s hard 6½-inch square boxes take a lot less room on the shelf. Given a box roughly half the thickness and half the height of some of the larger computer magazines, you can store a lot more magazines in the same space. Since they take less space, subscribers are more apt to keep them, which means less agony from realizing that a particular article went out in the trash two months ago because the garage filled up.

Although *Mentor* is informative and many of the programs included with it are undeniably useful, the magazine itself is visually unexciting. Advertisements are simple text blocks, and no large headers are used for the articles. Inclusion of a few simple graphics — even graphics built out of text characters to maintain compatibility with IBM Monochrome monitors — would go a long way towards brightening the magazine's image.

This lack of a good visual image is one that strikes right at the heart of advertising. Most magazines subsidize their publishing costs through sales of advertising, and major advertisers have traditionally been geared toward full-page color ads. It remains to be seen whether they will tackle display screen formats, although the concept does bring up some interesting opportunities for the future, when advertisements may consist of full-color animated sequences complete with sound.

One enhancement that *Mentor* definitely needs is some sort of hard-copy table of contents. At present the only way you can see what is in each issue is to mount it in your drive and take a look. Perhaps the *Mentor* people could include a large square label for subscribers to attach to each issue's box or simply print the table of contents on the diskette label.

Minimum system requirements for using *Mentor* are a 64K IBM PC system with a monochrome or color display and at least two single-sided drives. However, a special PCjr edition that requires only one drive is in preparation and may be available by summer of 1984. The *Mentor* staff expects to be publishing a good deal of PCjr software in the coming issues, including color graphics, animation, and entertainment programs.

The editors of *Mentor* have latched onto an idea that has almost limitless opportunities for software distribution, education, and entertainment. It will be interesting to watch this diskzine over the next couple of years to see how the editors and publishers handle their myriad challenges.

DDJ

by Ray Duncan

iRMX-86 for the IBM PC

Intel's iRMX-86 operating system for the 8086 series of microprocessors is a far cry from MS-DOS or CP/M-86. It is a real-time, multi-tasking, multi-user operating system with an excellent assembler and linker and an extensive arsenal of classy, optimizing compilers. The high-level languages available include PL/M-86, 87PASCAL, and 87FORTRAN—all support the "large memory model," reentrant code, direct control of the hardware . . . and, as you might have guessed from the names, all of the languages fully support the 8087 numeric coprocessor.

The Intel assembler, ASM86, is powerful, fast, and very polished. The macro facilities are solid and extensive; the Code-Macro facility is also present that allows you to add new opcodes to the assembler. A particularly nice feature is that you can link your programs with either a true 8087 function library or an 8087 emulator library without changing a word of the source code.

iRMX-86 has layers upon layers of interfaces and features. Your programs can talk to the operating system through calls to the Nucleus Basic I/O System, Extended I/O System, or UDI—each level has its own set of executive services and calling conventions. An additional layer, the Human Interface, contains the command utilities for formatting and copying disks, reading directories, managing files, and loading application software.

With all of this power, of course, comes considerable complexity. There are about a thousand pages of documentation or so just to cover the operating system proper, with plenty additional for the various language translators. Although the documentation is clear and well organized, it's heavily slanted toward the high-level language users, and the assembly language programmer will find it more difficult to extract the critical information necessary to get programs running. I wasted a full day just puzzling out the fact that you have to "attach" the console input stream but "create" the console output stream. The error messages that you get along the way from the system loader are cryptic to say the least.

iRMX has been independently ported to the IBM PC by two different vendors. It is available under the name "PC/iRMX" for $2,250.00 from Real-Time Computer Science Corp. (RTCS), P.O. Box 3000, Camarillo, CA 93011. It is sold under the name "RTOS" for $600.00 by Microware, P.O. Box 79, Kingston, MA 02364. Both companies include the assembler and linker with the operating system, but charge you extra for the various high-level languages.

These two products are definitely aimed at the pro and are not for the casual user. Although it probably only took you an hour or so to read the PC-DOS manual and get comfortable with the system, it will likely take you a week to become productive with iRMX-86. After porting Laboratory Microsystems PC/FORTH to run under both the RTCS and Microware implementations, I have conluded that I like the Intel operating system and development tools very much but that the IBM PC only barely has the horsepower to make them run properly. You certainly should think twice about buying iRMX unless your PC has a hard disk and the maximum amount of RAM.

Concerning Redirection

Chet Floyd of Manhattan Beach, California, writes: "Perhaps the good Doctor or a reader can explain a puzzling quirk of IBM's PC-DOS 2.00 in the redirection feature. File redirection works perfectly with DOS commands, but less perfectly with C and Pascal programs that use STDIN. The problem occurs when STDIN is redirected to a disk file because end-of-file is not sensed unless the EOF marker (^Z) is explicitly placed in the file. This can be easily done, of course, but having to do so is an inconvenience, particularly since DOS will hang if actual EOF occurs before the character is read. Contrary to the system documentation, ^C will not recover; the system must be rebooted.

"For example,

```
sort <text.fil
```

works swell, but

```
myprog <text.fil
```

hangs at the end of file unless ^Z is read as a character from the file.

"The IBM Pascal manual points out that STDIN never returns TRUE for EOF unless STDIN is redirected. But it seems not to work this way, and, as mentioned, C programs suffer the same way. Overcoming this problem would make the rudimentary piping and redirection that PC-DOS offers much more usable."

Any comments from the MS-DOS/PC-DOS wizards out there?

C Programming Tools

C-INDEX+ is a new data management software tool that provides full B+Tree ISAM indexing, variable length data storage, variable length keys, and "virtual memory management" of records. The package comes with an interactive tutorial, a detailed programming guide with examples, and a reference guide by function call. C compilers currently supported include Lattice C, Microsoft C, Computer Innovations C-86, and Manx Aztec CII. There is no license fee for application software that embeds C-INDEX+. Object code license is $400.00 and source code is $2000.00 from Trio

Systems, 2210 Wilshire Blvd., Suite 289, Santa Monica, CA 90403.

Kurt Klinzing, of Novum Organum, 29 Egerton Road, Arlington, MA 02174 was kind enough to send me a review copy of his company's product called "C Building Blocks." This is a beautifully documented, incredibly comprehensive set of C function libraries. The following modules can be purchased independently:

- C Building Blocks I (string functions, printer and serial port access, directory management, file management, operating system services, and video display) $149.00.

- Advanced Building Blocks (field directed input, window management, Julian date conversion, event timers, data compression). Requires C Building Blocks I also. $99.00.

- Mathematics Building Blocks (trig, logs, exponentials, random numbers) $99.99.

- Database Building Blocks (B-Tree indexes, direct or sequential access by key, variable length records) $149.00.

- Telecommunications Building Blocks (communications port control and character I/O, modem control including autodial, file transfer with Xmodem protocol) $149.00.

The C Building Blocks are written in Lattice C and include all source code. I've embarked on a rather large application using these libraries, so a more detailed review will be forthcoming in a later column.

Savage's Benchmark Again

This month we'll print one last source listing for a program that implements Bill Savage's floating-point benchmark, and next month we'll publish the revised, enlarged collection of results. Chris Dunford of Columbia, Maryland, writes: "I was interested in the floating-point performance/accuracy test results published in March and I thought I would see just how fast we could get an 8087-equipped IBM PC to run. The result is the attached 8088/8087 assembler program (see Listing, page 118) which executes in 2.2 seconds with an error of 3E-10 by your measure. I think this is pretty impressive. The accuracy beats everything in the March results chart except the IBM 3081, and the performance ranks tenth, ahead of such notables as HP9000, PDP-11, LSI-11, and some of the VAX timings. Here are a few notes on the program . . .

"The assembler used was version 1.3 of Digital Research's RASM-86, which supports 8087 operations (albeit with some non-standard mnemonics). Both RASM-86 and the code files produced by it run under PC-DOS. Clearly, the same results could have been achieved using the Microsoft Assembler with an appropriate set of macros; one such set is marketed by Southwestern Data Processing in Tucson. I chose RASM-86 simply on the basis of speed. Strangely, DRI's debugger, SID-86, does *not* support 8087 mnemonics even though the assembler does.

"In the interest of brevity, the program contains no code to display the final value of 'A'; a floating point-to-ASCII routine would have considerably lengthened the size of the listing (but not the execution time). I determined the 'A' value using a patched version of DEBUG which displays the 8087 register contents. Timings were obtained by adding a routine to count ticks of the system timer. This is accurate to within about 1/18th of a second, give or take a tick.

"Some readers may take exception to the fact that the algorithms employed are substantially optimized for the problem at hand. For example, there is no error checking (no errors are possible unless there is a logic flaw in the code), and there is no need for the tangent function to examine the octant of the given angle (they are all in the 45–90 degree octant). However, it's my assertion that this remains a valid comparison: one of the few remaining benefits of assembler is that such optimization is possible. That's what separates assembler code from compiler code—it's designed for the problem being solved and need not concern itself with correcting for situations that cannot possibly be encountered. In any event, I put together a version of the program with more generalized algorithms and found that the timings were substantially the same.

"I would point out that I am by no means an expert in 8087 programming, and I am certainly not a mathematician. It's quite likely that the program could be made more efficient by someone who is well-versed in either. Several of the algorithms in the program are perverted versions of those presented by Bill Rash in the Intel Application Note AP-113. I should also caution readers not to extract these algorithms for use in other programs unless they are looking forward to seeing a lot of unnormals, denormals, infinities of all shapes and sizes, and even the odd not-a-number. The routines were specifically written for this program and simply will not serve as general purpose transcendentals.

"While I was playing with all this, I also put together a test program for the p-System. The test runs under the IV.1 version as implemented by Network Consulting (version c1f). I turned off the range checking, used the 8087, and compiled to native code. Using double precision, the test ran in about 14 seconds and produced the astonishing error of 3E-10: the same error as the assembler program discussed above. My hunch is that this is artifact, but at least it is reproducible artifact. The timing, I think, is quite creditable and compares favorably with a number of the other compilers for the IBM machine. It compares extremely well with my famous brand C compiler, which took 86 seconds to produce a considerably less accurate result . . . And yes, the C compiler supports the 8087. All you 'p-System is too slow' people take note."

Thanks, Chris, for a very informative letter and program.

DDJ

(Listing begins on page 118)

16-Bit Listing (Text begins on page 116)

```
                        pagewidth 120
                        title ' DDJ Floating Point Test'
                        noiflist

                        ; ---------------------------------------------------------
                        ;                   FPTEST.A86     03/14/84
                        ; IBMPC/8087 DDJ Floating point performance/accuracy test
                        ;
                        ; By Christopher J. Dunford
                        ;    10057-2 Windstream Drive
                        ;    Columbia, Maryland 21044
                        ;    (301) 992-9371
                        ;
                        ; Portions adapted from:
                        ;   "Getting Started With the Numeric Data Processor"
                        ;   Bill Rash, Intel Applications Note AP-113
                        ;
                        ; Program performs the following C equivalent:
                        ;
                        ;   main()
                        ;   {
                        ;     unsigned i = 2499;
                        ;     double a = 1, tan(), atan(), exp(), log(), sqrt();
                        ;
                        ;     for ( ; i ; i--)
                        ;        a = tan(atan(exp(log(sqrt(a*a))))) + 1.0;
                        ;   }
                        ; ---------------------------------------------------------

                        cseg
                        main:

                        ; ----- Initialize 8088
0000 1E                         push ds                 ; Set up long ret to DOS
0001 2BC0                       sub ax,ax
0003 50                         push ax
0004 B80000       R             mov ax,data             ; Establish data addressability
0007 8ED8                       mov ds,ax
0009 B9C309                     mov cx,2499             ; Loop counter

                        ; ----- Initialize 8087 NDP
000C 90DBE3                     fninit                  ; NDP reset
000F 90D93E0000   R             fnstcw control          ; Set rounding mode to chop
0014 810E0000000C R             or control,0C00H
001A 9BD92E0000   R             fldcw control
001F 9BDB2E0400   R             fld80 half_a_pi         ; Load a constant pi/2
0024 9BDB2E0E00   R             fld80 qtr_pi            ; And a constant pi/4
0029 9BD9E8                     fld1                    ; Initialize A to 1
                        eject

                        ; ----- Main test loop begins here
                        bigloop:
002C 9BDCC8                     fmul st0,st0            ; ST = A*A
002F 9BD9FA                     fsqrt                   ; ST = sqrt(A*A)
```

```
                ; ----- Compute (ln(sqrt(A*A))) -------------------------
                ; The NDP does all of its logs and exponentials in base 2.
                ; Thus we must compute ln(x) indirectly using the identity
                ;         ln(x) = ln(2) * log2(x)
                ; Fortunately, ln(2) is available as an NDP constant.
                ; In this and all following, 'q' is the current value of A;
                ; at this point, q is sqrt(A*A).
0032 9BD9ED             fldln2              ; ST = ln(2)
0035 9BD9C9             fxch                ; ST = q; ST(1) = ln(2)
0038 9BD9F1             fyl2x               ; ST=ln(2)*log2(q) = ln(q)

                ; ----- Compute exp(ln(sqrt(a*a))) -----------------------------
                ; A bit tricky because the only exponential functions available are:
                ;    y = 2^x, where x is an integer (FSCALE), and
                ;    y = (2^x) - 1, where 0 <= x <= 0.5 (FY2XM1).
                ; We have to split the exponent into several parts and construct the
                ; final result from the partial results. Again, the NDP works only
                ; in base 2, so we must use the identity:
                ;           e^x = 2^(x*log2(e))
                ; The algorithm also uses the identity:
                ;           x^(y+z) = x^y * x^z
                ; In the following, f is the fractional part and i the integral part
                ; of q*log2(e).  See the reference for a similar routine (pp. 42-43).

003B 9BD9EA             fldl2e              ; Load constant log2(e)
003E 9BDEC9             fmul                ; ST = q*log2(e)
0041 9BD9E8             fld1
0044 9BD9E0             fchs                ; ST = -1
0047 9BD9C1             fld st1             ; ST = q*log2(e); ST(1) = -1; ST(2) = ST
004A 9BD9FC             frndint             ; ST = int(q*log2(e)) = i
004D 9BD9CA             fxch st2            ; ST = q*log2(e); ST(2) = i
0050 9BD8E2             fsub st,st2         ; ST = frac(q*log2(e) = f
0053 9BD9FD             fscale              ; ST = f/2  [f*(2^-1)]
0056 9BD9F0             f2xm1               ; ST = 2^(f/2) - 1
0059 9BDEE1             fsubr               ; ST = 2^(f/2);  ST(1) = i
005C 9BDCC8             fmul st0,st0        ; ST = 2^(f/2) * 2^(f/2) = 2^f
005F 9BD9FD             fscale              ; ST = 2^(q*log2(e)) = exp(q)
0062 9BDDD9             fstp st1            ; Dump one stack level
                eject

                ; ----- Compute atn(exp(ln(sqrt(a*a)))) ------------------
                ; The available function is ST = atn(ST1/ST), where
                ; ST(1)  < ST.  This implies that we must always take the
                ; atn of n where 0 < n < 1; however, the test requires
                ; atn of 1 <= n <= 2499.  Help is available in the form of
                ; atn(n) = pi/2 - atn(1/n).  We'll take the atn of 1/x and
                ; subtract the result from pi/2.

0065 9BD9E8             fld1                ; ST = 1; ST(1) = q
0068 9BD9C9             fxch                ; ST = q; ST(1) = 1
006B 9BD9F3             fpatan              ; ST = atn(1/q)
006E 9BD8E2             fsub st0,st2        ; ST = atn(1/q) - pi/2 = -atn(q)
0071 9BD9E0             fchs                ; ST = atn(q)
```

(Continued on next page)

16-Bit Listing (Listing Continued, text begins on page 116)

```
                        ; ----- Compute tan(atn(exp(ln(sqrt(a*a))))) ------------------
                        ; Too bizarre to describe here.  See the above reference, p.58ff.
                        ; The adaption is rather highly optimized for this test.

0074 9BD9F8                     fprem                   ; ST = q MOD pi/4
0077 9BD8E9                     fsubr st0,st1           ; ST = pi/4 - (q MOD pi/4)
007A 9BD9F2                     fptan                   ; Compute partial tangent
007D 9BDEF1                     fdivr                   ; Compute final tangent

                        ; ----- End of loop.  Add 1 to A and loop back
0080 9BD9E8                     fld1                    ; ST = 1;  ST(1) = A
0083 9BDEC1                     fadd                    ; ST = A + 1
0086 E2A4           002C        loop bigloop
0088 CB                         retf

                        ; ----- Data area ---------------------------------------------
                        dseg word
0000                    control   rw 1                                      ; NDP control
0002                    status    rw 1                                      ; NDP status
0004 35C26821A2DA       half_a_pi dw 0C235H, 02168H, 0DAA2H, 0C90FH, 03FFFH ; 80-bit pi/2
     0FC9FF3F
000E 35C26821A2DA       qtr_pi    dw 0C235H, 02168H, 0DAA2H, 0C90FH, 03FFEH ; 80-bit pi/4
     0FC9FE3F
                        end main

END OF ASSEMBLY.  NUMBER OF ERRORS:   0.  USE FACTOR:  0%
```

End Listing

OF INTEREST

Michael Wiesenberg

Give Me Your Phone Number

There are probably lots of vendors who wonder each month why their particular product or service goes unnoticed by this column. Let me explain in part how material for this column is filtered. *DDJ* receives literally hundreds of press releases each month. Before I even see the stack, someone usually culls announcements of the appointment of Sharon Apartment as Vice President of Marketing of Plastic Fantastic Diskettes and of the move of Hitek Analytic Engines from Palo Alto into their newly expanded facilities in Milpitas. I sift through the remaining dross for gems that I hope will be "Of Interest."

I almost automatically reject announcements of products unaccompanied by price because I figure if a company doesn't want to let people know how much its widget costs before a prospective customer sends an inquiry, the product must be overpriced. I also rarely use a product description sent by a company that doesn't list its phone number. I'm trying to look out for the readers, and I think a company that doesn't want phone calls has no intention of supporting its products. If I receive a press release that looks as if it were written by a grade school dropout, I also usually pitch it, my reasoning being that a company that can't take care in writing press releases probably does indecipherable documentation and is likely to take less care in the design of its products.

So there you have it. Maybe my conclusions are unwarranted, but that's how I operate. Companies that want attention should give me a phone number and tell me how much the product costs. They shouldn't attempt to hide the product's deficiencies behind a lot of overblown puffery. (I'm a technical writer by trade; I see through that stuff.) And if they can't write decent English, they should hire someone who can to write the release.

Don't Be Embarrassed in Spanish

El Ortografico, from Ibersoft (good name!), is the first-ever spelling checker for the Spanish language. It also checks for proper accents and lets you look up on-line the conjugation of most verbs. It also verifies your corrections as you enter them. You can use this speller with any word processor that stores text in ASCII, and you can customize for your word processor the unique characters of the language. If your word processor cannot represent the special characters, El Ortografico also has a utility program that permits printing them (if your printer is able to do so). The program runs on TRS-80 Models I, III, and 4, CP/M, PC-DOS, and MS-DOS, and will soon be available for Apple II. Sounds like a great product for $99.95.

Spell 1000 Times Faster

Since there seems to be interest in TEX and since it appears that a number of you have HP computers or use them on the job, I offer the following. JDJ Wordware offers **JSPEL/1000** for RTE-6/VM and RTE-A operating systems on HP 1000 computers. It features high-speed performance (in excess of 30,000 words per minute) by word caching. It generates almost instantaneously up to 10 correction candidates for what it considers an error. You merely enter the number of the correction you wish, and JSPEL makes the replacement. Each error is displayed in place in the line where it occurs, and you can generate a 20-line window of context around the suspect word by typing one letter.

JSPEL uses multiple dictionaries: a 27,000-word main dictionary that you may expand indefinitely, an incremental dictionary to store new words you wish to add to the main dictionary, and a customized dictionary that can be specific to a particular file—containing, for example, computer acronyms for a document about computers. JSPEL detects adjacent repeated words words. (It would have corrected that last sentence for me.) It can be used in look-up mode to find any words, and the search patterns can include any arbitrary selection of several kinds of wild cards. You TEX users will be pleased to know that JSPEL ignores TEX code sequences. The program supports the new hierarchical file system. A relocatable license costs $625, and source is $1500. You then can get software support for $20 a month and manual update service for $5 a month. The reference manual alone is $15.

TEXies Meet

The **TEX User's Group** will meet at Stanford August 13–24, during which two courses will be offered in the use of TEX: *Book Design Utilizing TEX* on the 13th and 14th, and *TEX for Beginners* August 20–24. Write to TUG care of the American Mathematical Society.

Plastic Seat Covers for Keyboards

Have you seen those tacky plastic seat covers people use in their homes when they don't want anyone to get their precious couches and chairs dirty? Now you can get them for your com-

puter's keyboard, in case you're the kind likely to spill coffee on it. **Safeskin** is molded to fit a particular keyboard like a glove and to remain in place during use. It has tactile "home-row" and numeric character locators and is made of antistatic polymer, through which you can see easily key tops and side markings. Merritt Computer Products sells them for IBM PC and compatibles, Apple IIe, TI Professional, TRS-80, Televideo, and Wang keyboards; they cost $29.95.

Z-100 Memory and Storage

PIICEON has a 256K memory board for Zenith Z-100 at $750; a 10Mb Winchester drive, the **ZD 100-10**, for $1795; and a 20Mb drive, the **ZD 100-20**, for $2285. All come with software and all necessary cables and connectors and are compatible with all levels of Z-DOS. PIICEON also has memory and disk upgrade kits for IBM PC and Alpha Microsystems.

Hang a PC on your Model 100

If you own a Radio Shack Model 100 computer (and sales figures for the little portable indicate that it can't be *just* computer-magazine writers who have been buying it), you might be interested in a program that lets the Model 100 use the disk storage of another computer. **Disk+** from the Portable Computer Support Group in Dallas is supplied in two pieces—part on cassette for the Model 100 and part on disk for the other computer: currently, the IBM PC and most MS-DOS machines, Radio Shack computers, Apple II, IIe, II+, Olivetti ETV 300, M20 and M24, and some CP/M computers. You select Disk+ from your Model 100 RAM file menu, and the menu turns *into* a disk menu showing the files on your other machine. You can then pull files into Model 100 RAM from disk or save RAM files to disk at up to 19200 baud; your large computer has become a peripheral for your Model 100. You can also create and manipulate disk subdirectories to handle groups of files at one time. Disk+ costs $69.95 and requires a serial cable with a null modem (unless you use the Model 100's internal modem; not recommended, since you'll be limited to 300 baud disk accesses). One caveat: this thing uses 9K of the Model 100's limited RAM.

Your China Trip

If you're planning to be in Xiamen toward the end of November you should check out **Computer China '84**, an exhibition introducing the latest micro and minicomputer technology to the people of the People's Republic. It seems that the personal computer revolution is catching fire in China, with grass-roots microcomputer associations and user's groups springing up in the provinces. There are only about 40,000 micros in China today, including single-board computers, but Adsale Exhibition Services, which is coordinating overseas vendors and exhibitors, expects a good turnout for the exposition-plus-conference.

Price Cut at the Co-op

CPMUG, the CP/M User's Group, has cut the prices for its volumes of CP/M software. In case you were unaware, CPMUG's been running a huge software exchange program for some time now, some 100 volumes of CP/M software of mixed quality, including everything from source-code interpreters and compilers to games and printer pictures. They sell it in bulk, like wheat germ at the co-op, on 8-inch IBM or 5¼-inch Kaypro double-density single-sided, Epson QX-10 double-density double-sided, Apple 16-sector, and North Star double- or quad-density disks. They cut the price from $13 to $10 for 8-inch disks and from $18 to $15 for 5¼-inch disks; that's for the U.S., Canada, and Mexico. They dropped the corresponding prices for other destinations from $17 to $13 and from $21 to $18, respectively. CPMUG will send you a catalog for $10 ($15 for you non-North Americans).

CP/M Calling CP/M

Softcom Telecommunications Utility for CP/M, from The Software Store, is a terminal emulator for mainframe time-sharing systems; it downloads files from a host system, sends text from your disk to, they claim, "almost any type of computer," and exchanges any type of file with other Softcom systems. The intelligent terminal mode transfers data at up to 9600 baud in full or half duplex and supports the XON/XOFF protocol. You need an 8080, 8085, or Z80 CP/M system, with at least 32K, and $150.

Half a Board Is Better Than . . .

One of the first in the race for add-ons for the IBM PC Portable is Ven-Tel with their **PC Modem Half Card**, a 1200/300 baud, auto-answer, auto-dial internal modem, for $549. Since the Portable has only half-sized expansion slots, only boards of this size fit inside the computer. Apparently modified from their Half Card modem for the PC-XT, the Half Card comes with CrossTalk-XVI, instructions, and a phone cable.

Hinkey Dinkey Disk-a-Do

If you have diskettes scattered all over the top of your desk gathering dust and soaking up electric fields, you need **Disk-a-Do** diskette storage units from Information Concepts. These rotating lazy-Susan units are made of molded black ABS *(what's that?)* supported on a steel base plate with ball bearings. They have stenciled slot index numbers, and the first slot has a printed index directory card. Model 60 holds 60 diskettes and costs $74.95. Model 60C holds 60 and has a bronze acrylic dust cover cabinet with a window toward which you rotate the unit until the dis-

kette you want appears; it costs $99.95. There's also a 120C that holds 120 diskettes and has the cover, but they neglected to tell us the price. Add $2 p&h per unit.

Where's the Widower?

I'm waiting for a lot of irate letters from feminists on this one. The **Computer Widow Tee Shirt** from Crabapple is, they say, "an ideal gift for that wife or girl friend who thinks you spend too much time computing. [It] features white high tech lettering on an all black fabric and is a perfect way to compensate for all those evenings you spent at the keyboard." $9.95 plus $1 p&h.

IBM Recalls Everything (Except IBM)

IBM (Incredibly Big Manufacturer) has announced, through its wholly owned subsidiary, the United States Government, a recall on all privately held personal computers, citing a defective connector that could disintegrate during normal operation, releasing deadly dimethyltryptamine gas. (Bonuses will be paid those turning in plug-compatible look-alikes.) Recalled computers will be replaced at no cost with IBM PCjrs that have been modified to accept only IBM software. (All so-called compatible software will self-destruct in these special machines.) Naturally these jrs are compatible with no other IBM products (or anything else). In other news, IBM is rumored to be planning a midyear fourfold increase in the prices of all PCjr software.

Contact Points

Adsale Exhibition Services, 21/F, Tung Wai Commercial Building, 109-111 Gloucester Road, Wanchai, Hong Kong; 5-8920511 (phone); 63109 ADSAP HX (Telex).

American Mathematical Society, Box 6248, Providence, RI 02940; (401) 272-9500, ext. 232.

Ariel Corporation, 600 West 116th St., New York, NY 10027; (212) 662-7324.

CP/M User's Group (CPMUG), 1651 Third Avenue, New York, NY 10028.

Crabapple, Inc., Box 3236, Framingham, MA 01701; (617) 877-9242.

Ibersoft, Box 3343, Trenton, NJ 08619; (609) 890-1496.

Information Concepts, Inc., Box 462, Stone Mountain, GA 30086; (404) 979-8479.

JDJ Wordware, Box 354, Cupertino, CA 95015; (415) 965-3245.

Merritt Computer Products, Inc., 2925 LBJ Fwy., Suite 180, Dallas, TX 75234; (214) 942-1142.

PIICEON, 2114 Ringwood Ave., San Jose, CA 95131; (408) 946-8030.

Portable Computer Support Group, 11035 Harry Hines Blvd., #207, Dallas, TX 75229; (214) 351-0564.

TEX User's Group: see American Mathematical Society.

The Software Store, 706 Chippewa Square, Marquette, MI 49855; (906) 228-7622.

Ven-Tel, Inc., 2342 Walsh Ave., Santa Clara, CA 95051; (408) 727-5721.

■■J

An Infinite Key Encryption System

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

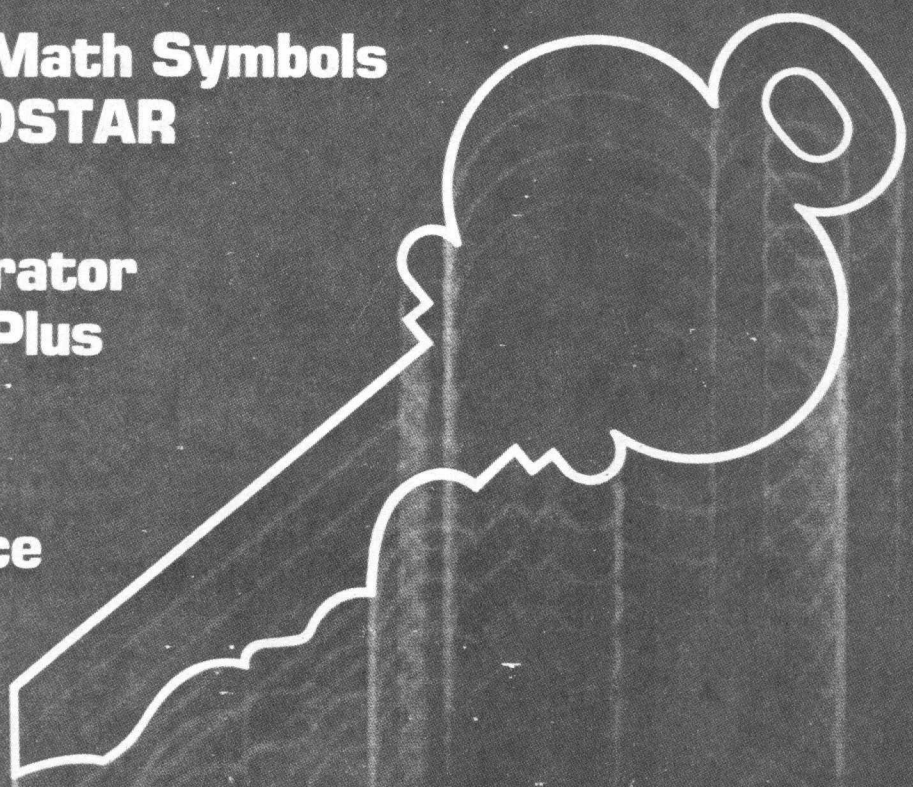
#94 August 1984

$2.95 (3.50 Canada)

SCISTAR
Greek and Math Symbols with WORDSTAR

File Comparator for CP/M Plus

A Small-C Concordance Generator



0 08
38351 16562

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*

Production
Design Director *Fred Fehlau*
Art Director *Shelley Rae Doeden*
Production Manager *Detta Penna*
Production Assistant *Alida Hinton*
Cover Artist *Claudia Majewski*

Advertising
Advertising Sales *Alice Hinton, Walter Andrzejewski*

Circulation
Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.
Publisher and Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

August 1984
Volume 9, Issue 8

CONTENTS

In this Issue

This month's articles focus primarily on text files, from protection to comparison to indexing to printing. Our cover article presents a public-domain, infinite-key encryption system for CP/M, with discussion ranging from the theory behind constructing a secure encryption system to practical considerations that can affect the design. For those interested in trying out (or trying to break) this system, the code and documentation should be available from a number of sources by the time you read this. In addition to obtaining a disk from the authors, a couple of places you should find it are on Byron McKay's PicoNet RCPM (415-965-4097) and Mel Cruts' RCPM (408-263-2588). We'll update you periodically on where else it may be obtained through the Letters to the Editor column.

Upcoming Special Issues

Next month we present our fourth annual Forth issue. As in past years, the response has been good, and you should find it useful and educational. Unix authors should note that we are planning a special Unix issue for this December. Those interested may contact us by phone or mail to discuss topics. We should receive manuscripts by September 7, 1984. We prefer receiving manuscripts via magnetic or electronic media, but we are still building our repertoire of capabilities. Contact us to coordinate disk formats or electronic services.

A Brief Reminder

We have already begun receiving entries for the Doctor's Fifth Generation Programming Competition. Those who wish to participate should remember that entries should be postmarked no later than September 31, 1984. If this is the first you have heard of it and would like details, give us a call or look in the May and June issues of *DDJ*.

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to remarks to the editors concerning accuracy and relevance on manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness. Their remarks help prevent authors from exposing blindspots or misconceptions in print and help ensure that our readers receive clear and accurate information. The referees who contributed to this month's issue are:

Wayne Chin, Hewlett-Packard
John P. Keyes, Syracuse University
John K. Taber, IBM

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

SCISTAR—Greek and Math Symbols with WordStar 26
by John Thomas and Guy Fletcher
Describes how to implement a universal set of user-programmed printer commands using WordStar with MailMerge

What's The Diff?—A File Comparator for CP/M Plus 30
by D.E. Cortesi
How to design and implement a file difference finder in Pascal that saves the differences between two versions as a script of ED commands

Designing a File Encryption System 44
by John A. Thomas and Joan Thersites
How to build a strong, practical infinite-key encryption system for CP/M, including discussion on background, theory, and design tradeoffs

A Small-C Concordance Generator 86
by John E. Staneff
Extending an internal sort capability by using word lists during the sort input phase, this program extracts desired words from an input file and produces an index to their source

DEPARTMENTS

Editorial 6

Letters 8

Dr. Dobb's Clinic 22
Transcendental Sources; Over-Submitting; An Old Bug; BASIC Precision, Again

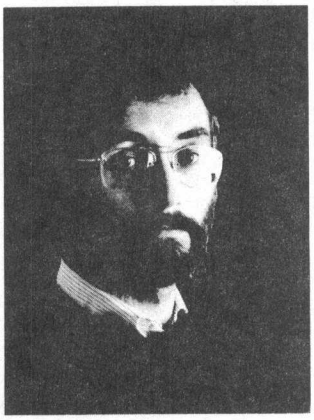
16-Bit Software Toolbox 106
More on Savage's benchmark: An updated table of results, some timings on spreadsheets and Modula-2, and a discussion on possible benchmark weakpoints

C/Unix Programmer's Notebook 116
uucp, public domain C, a standard fopen and fclose for BDSC, and some comments about C I/O and Unix comments

Software Reviews 122
PC-Write, PC Small-C 2.0

Of Interest 126

EDITORIAL



The Apple Macintosh is frustrating. Its major use right now seems to be for drawing pictures. Macwrite is slow. The amount of disk swapping the machine requires is unacceptable. The machine forces users to learn to operate a new control, the mouse. (MicroPro president Seymour Rubinstein says of the mouse that it's great for people with three hands.) And the Mac has no color.

If you write software or build hardware, the Mac is a source of particular frustration. To design software for it you have to buy a Lisa, and to buy a Lisa you have to sit on a rubber waiting list that always seems to be three months long. Hardware manufacturers can't do much with that closed box: you need a foot-long hex driver just to open the case, there are no slots and it takes courage to imagine modifying a four-layer motherboard. (Nevertheless, some nerveless hardware hackers are already creating their own 512K fat Macs.)

Despite all these frustrations, it remains the case that the Apple Macintosh has changed the entire nature of the game. It has redefined the *computer-person* interface, so radically altering the way people deal with computers that we can't go back to the simpler days. It makes the IBM PC look like a Processor Technology Sol.

Go into a computer store that has a Mac in stock and you'll see novices approaching and using the machine without fear or prodding. You'll hear people who claim not to like computers grudgingly admitting they like this one.

The Mac was designed with more thought for the user than any other personal computer. Consider the windows: there are window implementations for the IBM PC and other machines; why are they unsatisfying? Author Cary Lu thinks it's because they are still dependent on character-mapped graphics. The result: "the 'desktop' looks cluttered," according to Lu. The Mac's display design principle (everything is graphics) makes it simple to implement various character fonts, and the different fonts make the different window displays distinctive. So Mac windows are beautiful and natural, while windows on the PC look cluttered and kludgy.

Yes, the Mac lacks cursor keys, and the Model T lacked reins. Keyboards are good for typing letters and numbers. To imagine that they are the best way to represent movement in 2-space is to exhibit remarkable naivity.

Yes, MacWrite is slow. Some programmers recognize that inadequacy as an opportunity, and one of them will probably get rich from the observation. Better word processors will supplant the first version of MacWrite. At least two are in the works at Apple, but the eventual winner is likely to come from outside. Apple won't mind.

Apple (and, yes, Xerox PARC where the ideas originated) put considerably more thought into the Mac/Lisa user interface than MITS put into the Altair bus in 1974. But just as MITS's competitors took the Altair bus away from MITS and refined it into the S-100 bus, good programmers can make the Mac/Lisa user interface their own. Can, and will.

Probably just about the time some new development makes it obsolete.

Michael Swaine

Michael Swaine

LETTERS

A String Finding Function

Dear Editor:

I very much enjoyed the article "Optimizing Strings in C" by Edward McDermott in the April '84 issue (*DDJ* #90).

One of the several additional routines suggested was one for locating strings. I needed such a routine to run under the Software Toolwork's uptown sister of Small C, C/80. Since it passes its arguments on the stack in the same fashion that Mr. McDermott's listing appears to expect them, I thought that the accompanying Listing One (page 10) might be of value.

At the time it was developed, I needed a routine for locating a string in a CP/M file placed in RAM, thus the string to be searched is called a "Character file" and the default terminator is CP/M's end of file 1Ah. If EOF is defined otherwise in the program, any terminator such as 0 may be used.

Sincerely,
Paul C. Barton
Col. Wilkins Rd.
RD 1
Milford, NH 03055

Improved Directory

Dear *DDJ*:

I have enjoyed the Dan Daetwyler article "Sorted Diskette Directory for the IBM PC" (January *DDJ*, #87). One of the advantages of implementing something of that sort is the opportunity to add a few personal flourishes. I enclose two additions that do additional tasks, and a third which corrects an annoyance.

First, although one may not want hidden files included on the COVER, I do want to know that there are such files on the diskette, and how many. I have added code to count them and indicate the count at the end of the list.

Second, I find it very convenient to group like files together—e.g., list all .COM files first (alphabetically), then all .BAS files, and lastly all the other files. I have included a procedure (COORDR) that prompts for the file name extensions that you wish to place first on the sorted list, and changes in COSCAN and COPRNT that handle the modified sort and print.

Finally, I note that if you have a file with the maximum 8-character file name plus maximum 3-character extension, and a file length greater than 99999, there is no space in the listing between name and length. I have added a single space between each column to take care of this.

I enclose complete listings (Listing Two, page 11) of the added procedure COORDR and the two procedures which have significant additions (COSCAN and COPRNT, Listings Three and Four, pages 12 and 16). I also include full listing for COENDP (Listing Five, page 21) since, although there are only minor changes, it is short. In addition to the listings included, you need to add "GETORD:NEAR" to the EXTERN definition in the main program COVER, and the code line "CALL GETORD" following the "CALL GETTTL" (line 79 in the published listing of COVER). Finally, add code to make the PROC DECMAL in COTITL a PUBLIC definition.

Yours truly,
Bruce F. Cameron
4067 Rose Hill
Cincinnati, OH 45229

Relief for User Frustration

Dear Editor:

As you are no doubt aware, a problem of great dimension has surfaced in the computer industry, which is a cause of major concern for growing thousands of computer users. I speak of the problem of user frustration.

An individual who purchases a computer from a local dealer sometimes gets sold a "bill of goods." Even if the dealer has been completely honest and aboveboard, the average user faces the problem of climbing a veritable Everest of manuals and other documents in an attempt to put the new purchase to work. I think you will agree with me that a great percentage of such persons experience teeth-grating frustration.

What to do? I am proposing a practical solution to the problem. An article that appeared in the April 2 issue of *InfoWorld* described a "movement" I am attempting to organize. Calling ourselves the Society for the Prevention of Cruelty to Prospective Computer Purchasers (SPC–>PCP), our purpose is to link people with problems to people with solutions.

I have found through 20 years of experience in the computer industry that technical people are very friendly, on the whole, and willing to help others. I am trying to locate a large number of such technical people who would be happy to share their expertise with neophytes. Once I have accumulated such a list of experienced helpers, I shall advertise in computer magazines that such a list exists, and for a small fee (to cover the cost of the ads), anyone may send for a list of members in his or her immediate area.

The response I have been getting is overwhelmingly enthusiastic. The technical community is more than willing to support such an endeavor. In the process, of course, they may make relationships with some very important contacts. Everyone benefits!

I am hoping that, as a concerned member of the community of computer professionals, your magazine will lend support to SPC–>PCP by helping me to contact the experienced technical individuals needed to offset the wave of frustration that is growing larger with each passing day. Your assistance is urgently needed.

Thank you very much.

Yours very truly,
Burton Bhavisyat

SPC–>PCP
Rte 1, Box 318
Moundsville, WV 26041

DRI Indifferent?

Dear Editor:

I have never written a letter to *DDJ* before although I am a charter subscriber and I still have every issue since number 1. I have always admired your courage in publishing the facts to protect the public.

I feel it is necessary to warn users of an attitude at Digital Research which I will (with restraint) call "unabashedly indifferent." We used to accuse a particularly large mainframe manufacturer of this attitude, but it appears that Digital Research has fallen to new depths in the same attitude. Let me recount one small example, which will hopefully help others too.

We bought the Manx C compiler version 1.05G for $200. It had several bugs. We were promised version 1.06 would solve those bugs—unfortunately it cost us $250 to upgrade to version 1.06 *two months* after buying 1.05. That's bad enough. Version 1.06 also had bugs: some error messages had misspelled words in them and other very minor bugs.

The major problem was that the Manx linker crashed when trying to link a program which was correctly linked by 1.05. Version 1.06 advertised the ability to use the Digital Research RMAC/LINK programs. We said, great, we will use those. Apparently, no one at Manx had actually tried that—the compiler generated underscores and periods in names which the RMAC compiler could not handle. Manx has not yet communicated a fix for their linker, but did get me a patch after frequent calls (about six) and approximately two weeks. This did hold us up significantly in our development, but at least Manx did respond.

The patch was not for their compiler; instead, it was for RMAC. The patch, only a few lines, made RMAC accept underscores and periods. We discovered several bugs at that point with Digital Research's LINK—the worst was sprinkling of addresses on top of code when both ASEGs and CSEGs were used in the same area of memory. After we figured that out (it wasn't easy), we avoided the constructs which caused them. The problems could not be demonstrated with a small test program (apparently due to table overflows?), therefore we didn't bother DR, knowing the difficulties in dealing with them.

We finally came across a bug in RMAC (very difficult to find) which we traced back to the original nonpatched RMAC (version 1.1). The following program demonstrates it:

```
REPT    110
DB      0
ENDM
END
```

The DB statement gets repeated 78 times! After some experimentation we discovered that all numbers between 97 and 122 were reduced by 32. This, of course, in retrospect, is an unfortunate attempt by RMAC to convert lower case 'a' to 'z' to upper case. Since this statement is used frequently by the Manx C compiler, and the problem was so easily demonstrated, we decided to call Digital Research.

We subscribe to Digital Research's $250 software maintenance program because we learned long ago that their regular maintenance line is perpetually busy (I have an automatic dialer). To make a long story slightly shorter, after four days of phone calls, all the way to the head of marketing, their attitude seemed to be:

1. That DR doesn't support 8-bit products anymore, because they don't make DR money, and
2. That even if they decided to fix it, it would be a year or so.

What capped it off was that the head of marketing told us we might want to purchase the $250 maintenance package. That was the straw that broke the camel's back. When we informed him we currently did subscribe, he said, "Oh, maybe we can fix it in a year, but don't count on it."

In summary, Manx is a relatively small company. Their attitude is understandable, they aren't millionaires, and they do try. Digital Research seems to have gotten fat and sassy and said to heck with the user and his problems. A bug is a bug and should be fixed.

This is only one of dozens of problems I have had with DR since I started using CPM V1.3 in 1976. I have implemented the BIOS/XIOS for V1.3, V1.4, V2.2, V3.0 and MPM V1.1, V2.1, found many bugs, never had any of them fixed or replied to, written many letters to DR, and received *no* replies.

This was such a simple bug that caused so much trouble, I just had to write to warn potential DR users what I think they can expect.

Sincerely,
Steve Conley
2030 Powers Ferry Rd.
Suite 110
Atlanta, GA 30339 ■■J

Letters

Listing One

```
/*      The function char *findstr(is, cf) returns a pointer to the first
        occurence of string pointed to by is in that pointed to by cf.

        In the default case, IS is terminated by a 0 byte
        and CF is terminated by CPM's end of file 26 (^Z).

        To change these defaults, the calling program must
        define the constants EOS (end of string for is) and
                             EOF (end of file for cf).
```

```
#ifndef EOS                                                          */
#define EOS 0
#endif
#ifndef EOF
#define EOF 26
#endif

/******************/

char *findstr(is, cf)
char *is, *cf;
{
#asm
                POP B           ;Return address
                POP D           ;Character file -cf
                POP H           ;Index string - is
;
                PUSH H          ;Restore the stack for caller
                PUSH D
                PUSH B
;
SEARCH:         LDAX  D         ;*CF to A
SEARCH1:        CMP   M         ;*CF == IS[0] ?
                CZ    INDEX     ;If yes call (check rest of IS)
                INX   D         ;No match ++CF
                LDAX  D         ;Check for end of CF
                CPI   EOF
                JNZ   SEARCH1   ;Loop if not
FAIL:           LXI   D,0000    ;Search failed.  Load error message.
                JMP   DONE      ;And go back to caller
;
INDEX:          PUSH  D         ;Save registers
                PUSH  H
INDEX1:         INX   D         ;Next character.  We know the last one matched
                INX   H
                MVI   A,EOS     ;Check for end of IS
                CMP   M
                JZ    SUCCESS   ;If found match is complete.
                LDAX  D         ;Not yet
                CMP   M
                JZ    INDEX1    ;This character matched. Loop for next.
                POP   H         ;No match.  Restore registers
                POP   D         ;in preparation for
                RET             ;return to search of CF for match of IS[0]

;
;
SUCCESS:        POP   H
                POP   D         ;Restore pointer to first position of IS in CF
                POP   B         ;Pop call from search. Now will ret to caller
DONE:           XCHG            ;Pointer into HL for return.
#endasm
}
```

End Listing One

Listing Two

```
                        PAGE    ,132
                        TITLE   COORDR  -  Diskette contents list - Redefine Order
                        COMMENT                 * Added PROC - Feb 1984 *
                        PAGE    82
                ;
                ;       Request file extension entries which are to be sorted to the beginning of the list
                ;
0000            CODE    SEGMENT PARA PUBLIC 'CODE'
                        ASSUME  CS:CODE, DS:CODE
                ;
```

(Continued on next page)

Letters (Listing continued, text begins on page 8)

Listing Two

```
                                        EXTRN   RELOC:BYTE
0000  80                        SPCHAR  DB      80H                     ; Ordering character
0001  08 00                     XBUF    DB      8,0                     ; Input buffer for extension
0003     08 [                           DB      8 DUP (?)
               ??
                  ]

000B  0D 0A 45 6E 74 65         XPRMT1  DB      13,10,'Enter file name extension to relocate$'
      72 20 66 69 6C 65
      20 6E 61 6D 65 20
      65 78 74 65 6E 73
      69 6F 6E 20 74 6F
      20 72 65 6C 6F 63
      61 74 65 24
0033  0D 0A 20 20 20 74         XPRMT2  DB      13,10,'   to top of list ("\" to quit): $'
      6F 20 74 6F 70 20
      6F 66 20 6C 69 73
      74 20 28 22 5C 22
      20 74 6F 20 71 75
      69 74 29 3A 20 24
                                ;
                                        PUBLIC  GETORD
0057                            GETORD  PROC    NEAR
0057  BD 0000                           MOV     BP,0                    ; Counter of entries
005A  BF 0000 E                         MOV     DI,OFFSET RELOC
005D  BA 000B R                 LOOP:   MOV     DX,OFFSET XPRMT1        ; Prompt for extension
0060  B4 09                             MOV     AH,9
0062  CD 21                             INT     21H
0064  BA 0033 R                         MOV     DX,OFFSET XPRMT2
0067  B4 09                             MOV     AH,9
0069  CD 21                             INT     21H
006B  BA 0001 R                         MOV     DX,OFFSET XBUF
006E  B4 0A                             MOV     AH,0AH                  ; Get extension
0070  CD 21                             INT     21H
0072  80 3E 0003 R 5C                   CMP     XBUF+2,5CH              ; Check for done (\)
0077  74 32                             JE      QUIT
0079  8A 1E 0002 R                      MOV     BL,XBUF+1               ; Get length
007D  80 FB 02                          CMP     BL,2
0080  7F 13                             JNLE    STOR
0082  32 FF                             XOR     BH,BH
0084  8B CB                             MOV     CX,BX
0086  83 E9 03                          SUB     CX,3
0089  F7 D9                             NEG     CX                      ; Set loop count (3 - length of string)
008B  57                                PUSH    DI
008C  8D BF 0003 R                      LEA     DI,XBUF[BX+2]           ; Points to char just after string
0090  B0 20                             MOV     AL,' '                  ; Fill with blanks
0092  F3/ AA                            REP     STOSB                   ;   to 3 characters
0094  5F                                POP     DI
0095  B9 0003                 STOR:   MOV     CX,3
0098  BE 0003 R                         MOV     SI,OFFSET XBUF+2
009B  F3/ A4                            REP     MOVSB                   ; Extension from buffer to list
009D  A0 0000 R                         MOV     AL,SPCHAR
00A0  AA                                STOSB                           ; Add 'ordering' character
00A1  FE 06 0000 R                      INC     SPCHAR
00A5  45                                INC     BP                      ; Count entry
00A6  83 FD 0A                          CMP     BP,10
00A9  7C B2                             JL      LOOP                    ; Cycle if 9 or fewer entries
00AB  C6 05 00                  QUIT:   MOV     BYTE PTR [DI],0         ; Terminator to end-of-list
00AE  C6 06 0000 R 80                   MOV     SPCHAR,80H              ; Reset ordering character
00B3  C3                                RET
00B4                            GETORD  ENDP
                                ;
00B4                            CODE    ENDS
                                ;
                                        END

Segments and groups:

                N a m e                 Size    align   combine class

CODE . . . . . .                        00B4    PARA    PUBLIC  'CODE'

Symbols:
```

```
                N a m e                 Type    Value   Attr

GETORD . . . . . .              N PROC  0057    CODE    Global  Length =005D
LOOP . . . . . .                L NEAR  005D    CODE
QUIT . . . . . .                L NEAR  00AB    CODE
RELOC. . . . . .                V BYTE  0000    CODE    External
SPCHAR . . . . . .              L BYTE  0000    CODE
STOR . . . . . .                L NEAR  0095    CODE
XBUF . . . . . .                L BYTE  0001    CODE
XPRMT1 . . . . . .              L BYTE  000B    CODE
XPRMT2 . . . . . .              L BYTE  0033    CODE

Warning Severe
Errors  Errors
0       0
```

End Listing Two

Listing Three

```
                                        PAGE    ,132
                                        TITLE   COSCAN  -  Diskette contents list  -  Scan Directory
                                        COMMENT         * Version 1.0 - June 1983
                                                          Modified  Jan 1984 *
                                        PAGE    82
                                ;
0000                            CODE    SEGMENT PARA PUBLIC 'CODE'
                                        ASSUME  CS:CODE, DS:CODE
                                ;
                                        EXTRN   RELOC:BYTE, PNTR:WORD, SRCE:BYTE
                                ;
                                        PUBLIC  HIDNFL
0000    03 [                    HIDNFL  DB      3 DUP (0)               ; Working buffer for hidden file count
            00
                ]

0003  FF 00 00 00 00 00         EXTND   DB      0FFH,0,0,0,0,0          ; Dummy FCB for file find (extended)
0009  00                        ATRB    DB      0                       ; File attribute byte
000A  00 3F 3F 3F 3F 3F                 DB      0,'???????????'         ; Dummy FCB for file find (normal)
      3F 3F 3F 3F 3F 3F
0016     18 [                           DB      24 DUP (?)
              ??
                ]

                                ;
                                        PUBLIC  SCAN
002E                            SCAN    PROC    NEAR
002E  BF 0000 E                         MOV     DI,OFFSET PNTR
0031  33 C0                             XOR     AX,AX
0033  B9 0078                           MOV     CX,120
0036  F3/ AB                            REP     STOSW                   ; Clear pointer table
0038  BB 0000 E                         MOV     BX,OFFSET PNTR          ; BX to pointer list
003B  BF 0000 E                         MOV     DI,OFFSET SRCE          ; DI to file name stack
003E  33 C9                             XOR     CX,CX
0040  BA 0003 R                         MOV     DX,OFFSET EXTND         ; Search FCB
0043  C6 06 0009 R 02                   MOV     ATRB,2                  ; Set for hidden files
0048  EB 19                             JMP     SHORT MAIN
004A  C6 06 0009 R 04           SYS:    MOV     ATRB,4                  ; Set for system files
004F  EB 12                             JMP     SHORT MAIN
0051  C6 06 0009 R 06           IBM:    MOV     ATRB,6                  ; Set for IBMBIO, IBMDOS,...
0056  EB 0B                             JMP     SHORT MAIN
0058  88 0E 0000 R              FIL:    MOV     HIDNFL,CL               ; Save hidden file count
005C  33 C9                             XOR     CX,CX
005E  C6 06 0009 R 00                   MOV     ATRB,0                  ; Set for regular files
0063  B4 11                     MAIN:   MOV     AH,11H
0065  CD 21                             INT     21H                     ; Get first search entry
0067  EB 07                             JMP     SHORT INNER
0069  BA 0003 R                 LOOP:   MOV     DX,OFFSET EXTND
006C  B4 12                             MOV     AH,12H
006E  CD 21                             INT     21H                     ; Get next entry
0070  0A C0                     INNER:  OR      AL,AL
0072  75 10                             JNZ     DONE
0074  E8 0097 R                         CALL    SAVE                    ; File name to stack
0077  A0 0009 R                         MOV     AL,ATRB                 ; Fetch attribute of search
007A  BE 0093                           MOV     SI,93H                  ; Point to DTA attribute field
007D  3A 04                             CMP     AL,BYTE PTR [SI]
007F  75 E8                             JNE     LOOP                    ; Don't count if not requested type
0081  41                                INC     CX                      ; Count entries
0082  EB E5                             JMP     LOOP
0084  80 3E 0009 R 00           DONE:   CMP     ATRB,0                  ; If zero, all file types checked
0089  74 0B                             JZ      ALLFL
```

(Continued on next page)

Letters (Listing continued, text begins on page 8)

Listing Three

```
008B  80 3E 0009 R 04                   CMP     ATRB,4
0090  72 B8                             JB      SYS                     ; =2, next to system
0092  74 BD                             JE      IBM                     ; =4, next to IBM...
0094  77 C2                             JA      FIL                     ; =6, next to regular
0096  C3                       ALLFL:  RET                             ; Returns count in CX
0097                           SCAN    ENDP
                               ;
0097                           SAVE    PROC    NEAR
0097  80 3E 0009 R 00                   CMP     ATRB,0
009C  75 40                             JNZ     PASS                    ; Hidden file count, no name save
009E  51                                PUSH    CX
009F  89 3F                             MOV     WORD PTR [BX],DI        ; Save pointer to entry stack
00A1  83 C3 02                          ADD     BX,2                    ;   and step pointer table
00A4  57                                PUSH    DI                      ; Save address for ordering character
00A5  47                                INC     DI                      ; Step to start of name
00A6  BE 0088                           MOV     SI,88H                  ; Point to DTA file name
00A9  B9 0008                           MOV     CX,8
00AC  8A 04                     SVLP:   MOV     AL,BYTE PTR [SI]
00AE  3C 20                             CMP     AL,' '
00B0  74 06                             JZ      NMDNE                   ; End of name
00B2  88 05                             MOV     BYTE PTR [DI],AL
00B4  46                                INC     SI
00B5  47                                INC     DI
00B6  E2 F4                             LOOP    SVLP
00B8  BE 0090                   NMDNE:  MOV     SI,90H                  ; Point to DTA type field
00BB  EB 22 90                          JMP     TESTRE                  ; Check if relocation requested
00BE  80 3C 20                  STOTYP: CMP     BYTE PTR [SI],' '
00C1  74 09                             JZ      ALLDNE                  ; No file type
00C3  C6 05 2E                          MOV     BYTE PTR [DI],'.'
00C6  47                                INC     DI
00C7  B9 0003                           MOV     CX,3
00CA  F3/ A4                            REP     MOVSB                   ; Move type field to stack
00CC  C6 05 00                  ALLDNE: MOV     BYTE PTR [DI],0         ; Mark end of string
00CF  47                                INC     DI
00D0  BE 00A4                           MOV     SI,0A4H                 ; Point to file size
00D3  B9 0004                           MOV     CX,4
00D6  F3/ A4                            REP     MOVSB                   ;   and save in stack
00D8  58                                POP     AX                      ; 'Ordering' character
00D9  5D                                POP     BP                      ; Location for same (PUSH from DI)
00DA  88 46 00                          MOV     [BP],AL                 ; Put at start of string
00DD  59                                POP     CX
00DE  C3                        PASS:   RET
                               ;
00DF  57                        TESTRE: PUSH    DI
00E0  B0 A0                             MOV     AL,0A0H                 ; Default order high
00E2  BF 0000 E                         MOV     DI,OFFSET RELOC
00E5  80 3D 00                  TLP:    CMP     BYTE PTR [DI],0         ; End of reorder table
00E8  74 13                             JE      NOMTCH
00EA  57                                PUSH    DI                      ; Save start of string in table
00EB  56                                PUSH    SI                      ;    and FCB location pointer
00EC  B9 0003                           MOV     CX,3
00EF  F3/ A6                            REPE    CMPSB                   ; File extension to table
00F1  5E                                POP     SI
00F2  5F                                POP     DI
00F3  74 05                             JE      MATCH
00F5  83 C7 04                          ADD     DI,4                    ; Point to next entry
00F8  EB EB                             JMP     TLP
00FA  8A 45 03                  MATCH:  MOV     AL,[DI+3]               ; Reorder character to AL
00FD  5F                        NOMTCH: POP     DI                      ; Restore to value at entry
00FE  50                                PUSH    AX                      ; Reorder char to stack
00FF  EB BD                             JMP     STOTYP
                               ;
0101                           SAVE    ENDP
                               ;
0101                           CODE    ENDS
                               ;
                                       END

Segments and groups:

                N a m e                 Size    align   combine class

CODE . . . . . . . . . . . . . .        0101    PARA    PUBLIC  'CODE'
```

```
Symbols:

                N a m e                 Type    Value   Attr

ALLDNE . . . . . .              L NEAR  00CC    CODE
ALLFL. . . . . .                L NEAR  0096    CODE
ATRB . . . . . .                L BYTE  0009    CODE
DONE . . . . . .                L NEAR  0084    CODE
EXTND. . . . . .                L BYTE  0003    CODE
FIL. . . . . .                  L NEAR  0058    CODE
HIDNFL . . . . . .              L BYTE  0000    CODE    Global  Length =0003
IBM. . . . . .                  L NEAR  0051    CODE
INNER. . . . . .                L NEAR  0070    CODE
LOOP . . . . . .                L NEAR  0069    CODE
MAIN . . . . . .                L NEAR  0063    CODE
MATCH. . . . . .                L NEAR  00FA    CODE
NMDNE. . . . . .                L NEAR  00B8    CODE
NOMTCH . . . . . .              L NEAR  00FD    CODE
PASS . . . . . .                L NEAR  00DE    CODE
PNTR . . . . . .                V WORD  0000    CODE    External
RELOC. . . . . .                V BYTE  0000    CODE    External
SAVE . . . . . .                N PROC  0097    CODE    Length =006A
SCAN . . . . . .                N PROC  002E    CODE    Global  Length =0069
SRCE . . . . . .                V BYTE  0000    CODE    External
STOTYP . . . . . .              L NEAR  00BE    CODE
SVLP . . . . . .                L NEAR  00AC    CODE
SYS. . . . . .                  L NEAR  004A    CODE
TESTRE . . . . . .              L NEAR  00DF    CODE
TLP. . . . . .                  L NEAR  00E5    CODE

Warning Severe
Errors  Errors
0       0
```

End Listing Three

Listing Four

```
                                PAGE    ,132
                                TITLE   COPRNT  -  Diskette contents list  -  Print Cover Sheet
                                COMMENT         * Version 1.0 - June 1983
                                                  Modified  Jan 1984 *
                                PAGE    82
                        ;
0000                    CODE    SEGMENT PARA PUBLIC 'CODE'
                                ASSUME  CS:CODE, DS:CODE
                        ;
                                EXTRN   PNTR:WORD, STKCNT:WORD, TITLX:BYTE, HIDNFL:BYTE
                        ;
                                EXTRN   CONVRT:NEAR, DECMAL:NEAR
                        ;
                                PUBLIC  PSX, RESTR
0000  00                PSX     DB      0                       ; Pass counter
0001  0C 00             RESTR   DB      12,0                    ; Printer 'restore' forms
0003     07 [           DBUF    DB      7 DUP (0)               ; Work buffer for file size
               00
                  ]

000A  48 69 64 64 65 6E HIDF    DB      'Hidden files  ',0
      20 66 69 6C 65 73
      20 20 00
0019  20 20 00          DBLK    DB      '  ',0                  ; Double blank between columns
001C  7C 20 20 00       LFTB    DB      '¦  ',0                 ; Left border
0020  20 20 7C          RGTB    DB      '  ¦'                   ; Right border (include CR/LF)
0023  0D 0A 00          CRLF    DB      13,10,0
0026  00                BCNT    DB      0                       ; Body line counter
                        ;
                                PUBLIC  PRINT
0027                    PRINT   PROC    NEAR
0027  FE 06 0000 R             INC     PSX                     ; Count number of prints
002B  C6 06 0026 R 21          MOV     BCNT,33                 ; Set body line counter
0030  A1 0000 E                MOV     AX,STKCNT               ; Load entry count
0033  B6 04                    MOV     DH,4
0035  F6 F6                    DIV     DH                      ; Divide by number of columns
0037  0A E4                    OR      AH,AH
0039  74 03                    JZ      SETCNT                  ; Evenly divisible
003B  FE C0                    INC     AL                      ; Ragged edge
003D  98                       CBW
003E  50                SETCNT: PUSH    AX                      ; Entries per column count
003F  E8 014D R                CALL    DOBRDR                  ; Do upper border
```

(Continued on page 18)

Listing Four

```
0042  E8 013E R                    CALL    DOBLNE                  ;   and a blank line
0045  E8 015C R                    CALL    DOLFTM                  ; Do left margin
0048  BA 0000 E                    MOV     DX,OFFSET TITLX
0048  E8 00A5 R                    CALL    DOPRT                   ; Output the title line
004E  B9 0004                      MOV     CX,4
0051  E8 016E R                    CALL    CLER                    ; Add 4 spaces
0054  E8 0165 R                    CALL    DORGTM                  ; Do right margin
0057  E8 013E R                    CALL    DOBLNE                  ;   and another blank line
005A  59                           POP     CX

005B  8B E9                        MOV     BP,CX
005D  D1 E5                        SHL     BP,1                    ; Offset per column in pointer list
005F  BE 0000 E                    MOV     SI,OFFSET PNTR          ; Point to start of pointer list
0062  E8 015C R              OTLP: CALL    DOLFTM                  ; Do a left margin
0065  B2 04                        MOV     DL,4                    ; Set inner loop count to column #
0067  33 DB                        XOR     BX,BX                   ; Clear column offset reg
0069  E8 00B9 R              INLP: CALL    PRTENT                  ; Print stack entry
006C  03 DD                        ADD     BX,BP                   ; Step to next column entry
006E  FE CA                        DEC     DL
0070  75 F7                        JNZ     INLP                    ; End of inner loop
0072  E8 0165 R                    CALL    DORGTM                  ; Do a right margin
0075  83 C6 02                     ADD     SI,2                    ; Step to next pointer
0078  FE 0E 0026 R                 DEC     BCNT                    ; Decrement body line count
007C  E2 E4                        LOOP    OTLP                    ; End of outer loop
007E  32 C0                        XOR     AL,AL
0080  0A 06 0000 E                 OR      AL,HIDNFL               ; Check for any hidden files
0084  74 08                        JZ      BLNK                    ; No
0086  E8 0108 R                    CALL    PRHID                   ; Print hidden files count,
0089  80 2E 0026 R 02              SUB     BCNT,2                  ;    uses two lines
008E  8A 0E 0026 R           BLNK: MOV     CL,BCNT                 ; Load remaining body lines
0092  32 ED                        XOR     CH,CH
0094  E3 05                        JCXZ    NOFILL                  ; All used
0096  E8 013E R              FILL: CALL    DOBLNE                  ; Fill out body lines
0099  E2 FB                        LOOP    FILL
009B  E8 014D R              NOFILL: CALL  DOBRDR                  ; Do bottom border
009E  BA 0001 R                    MOV     DX,OFFSET RESTR
00A1  E8 00A5 R                    CALL    DOPRT                   ; Restore page
00A4  C3                           RET
00A5                         PRINT ENDP
                             ;
                                   PUBLIC  DOPRT
00A5                         DOPRT PROC    NEAR                    ; This subroutine prints string
00A5  52                           PUSH    DX                      ;  (pointer in DX on entry).
00A6  56                           PUSH    SI                      ; String terminated by a null byte
00A7  8B F2                        MOV     SI,DX
00A9  B4 05                        MOV     AH,5                    ; Print line function
00AB  8A 14                  DPLP: MOV     DL,BYTE PTR [SI]
00AD  0A D2                        OR      DL,DL
00AF  74 05                        JZ      PRTEND
00B1  CD 21                        INT     21H
00B3  46                           INC     SI
00B4  EB F5                        JMP     DPLP
00B6  5E                     PRTEND: POP   SI
00B7  5A                           POP     DX
00B8  C3                           RET
00B9                         DOPRT ENDP
                             ;
00B9                         PRTENT PROC   NEAR                    ; Print one stack entry
00B9  51                           PUSH    CX
00BA  52                           PUSH    DX
00BB  B9 000C                      MOV     CX,12
00BE  8B 38                        MOV     DI,WORD PTR [SI+BX]     ; DI points to stack entry
00C0  0B FF                        OR      DI,DI                   ; If entry is zero, blank space
00C2  74 37                        JZ      BLNK1
00C4  47                           INC     DI                      ; Step over reorder character
00C5  B4 05                        MOV     AH,5
00C7  8A 15                  PELP: MOV     DL,BYTE PTR [DI]        ; Print to the end of the
00C9  0A D2                        OR      DL,DL                   ;   name/type entry
00CB  74 36                        JZ      BLNK2                   ;   blank rest of 12 chars
00CD  CD 21                        INT     21H
00CF  47                           INC     DI
00D0  E2 F5                        LOOP    PELP
00D2  B9 0001                BACK: MOV     CX,1
00D5  E8 016E R                    CALL    CLER                    ; Insert 1 space
00D8  47                           INC     DI
```

```
00D9  8B 05                     MOV     AX,WORD PTR [DI]        ; Load file size
00DB  8B 55 02                  MOV     DX,WORD PTR [DI+2]
00DE  56                        PUSH    SI
00DF  BF 0003 R                 MOV     DI,OFFSET DBUF
00E2  E8 0000 E                 CALL    CONVRT                  ; Size to ASCII
00E5  5E                        POP     SI
00E6  BA 0003 R                 MOV     DX,OFFSET DBUF
00E9  E8 00A5 R                 CALL    DOPRT                   ; Print the size
00EC  5A                GONE:   POP     DX
00ED  52                        PUSH    DX                      ; Reload entry value
00EE  FE CA                     DEC     DL
00F0  74 06                     JZ      PUNT                    ; If last column don't space over
00F2  BA 0019 R                 MOV     DX,OFFSET DBLK
00F5  E8 00A5 R                 CALL    DOPRT                   ; Print 2 blanks between cols
00F8  5A                PUNT:   POP     DX
00F9  59                        POP     CX
00FA  C3                        RET
00FB  B9 0013           BLNK1:  MOV     CX,19                   ; No entry, blank entire column
00FE  E8 016E R                 CALL    CLER
0101  EB E9                     JMP     GONE
0103  E8 016E R         BLNK2:  CALL    CLER                    ; Blank remainder of field
0106  EB CA                     JMP     BACK
0108                    PRTENT  ENDP
                        ;
0108                    PRHID   PROC    NEAR
0108  E8 013E R                 CALL    DOBLNE                  ; Blank line
010B  E8 015C R                 CALL    DOLFTM                  ; Left margin
010E  B9 0014                   MOV     CX,20                   ; Twenty blanks
0111  E8 016E R                 CALL    CLER
0114  BA 000A R                 MOV     DX,OFFSET HIDF
0117  E8 00A5 R                 CALL    DOPRT                   ; Text
011A  A0 0000 E                 MOV     AL,HIDNFL
011D  BF 0000 E                 MOV     DI,OFFSET HIDNFL
0120  8B D7                     MOV     DX,DI                   ; Set address for print (below)
0122  E8 0000 E                 CALL    DECMAL                  ; Number of files to ASCII
0125  80 3E 0000 E 30           CMP     HIDNFL,'0'
012A  75 05                     JNE     TWOCH
012C  C6 06 0000 E 20           MOV     HIDNFL,' '              ; Zero suppress
0131  E8 00A5 R         TWOCH:  CALL    DOPRT
0134  B9 002E                   MOV     CX,46
0137  E8 016E R                 CALL    CLER                    ; Fill with blanks
013A  E8 0165 R                 CALL    DORGTM                  ; Right margin
013D  C3                        RET
013E                    PRHID   ENDP
                        ;
013E                    DOBLNE  PROC    NEAR
013E  51                        PUSH    CX
013F  E8 015C R                 CALL    DOLFTM                  ; Output a bordered blank line
0142  B9 0052                   MOV     CX,82
0145  E8 016E R                 CALL    CLER
0148  E8 0165 R                 CALL    DORGTM
014B  59                        POP     CX
014C  C3                        RET
014D                    DOBLNE  ENDP
                        ;
014D                    DOBRDR  PROC    NEAR
014D  B9 0058                   MOV     CX,88
0150  B2 2D                     MOV     DL,'-'
0152  E8 0170 R                 CALL    DLFIL                   ; Output a top or bottom border
0155  BA 0023 R                 MOV     DX,OFFSET CRLF
0158  E8 00A5 R                 CALL    DOPRT
015B  C3                        RET
015C                    DOBRDR  ENDP
                        ;
015C                    DOLFTM  PROC    NEAR                    ; Output "¦     "
015C  52                        PUSH    DX
015D  BA 001C R                 MOV     DX,OFFSET LFTB
0160  E8 00A5 R                 CALL    DOPRT
0163  5A                        POP     DX
0164  C3                        RET
0165                    DOLFTM  ENDP
                        ;
0165                    DORGTM  PROC    NEAR                    ; Output "    ¦"
0165  52                        PUSH    DX
0166  BA 0020 R                 MOV     DX,OFFSET RGTB
0169  E8 00A5 R                 CALL    DOPRT
016C  5A                        POP     DX
```

(Continued on next page)

Listing Four

```
016D  C3                        RET
016E                    DORGTM  ENDP
                        ;
016E                    CLER    PROC    NEAR                    ; Output CX blanks
016E  B2 20                     MOV     DL,' '
0170  B4 05             DLFIL:  MOV     AH,5                    ; Output DL character CX times
0172  CD 21             CLRLP:  INT     21H
0174  E2 FC                     LOOP    CLRLP
0176  C3                        RET
0177                    CLER    ENDP
                        ;
0177                    CODE    ENDS
                        ;
                                END

Segments and groups:

                N a m e                 Size    align   combine class

CODE . . . . . .                0177    PARA    PUBLIC  'CODE'

Symbols:

                N a m e                 Type    Value   Attr

BACK . . . . . .                L NEAR  00D2    CODE
BCNT . . . . . .                L BYTE  0026    CODE
BLNK . . . . . .                L NEAR  008E    CODE
BLNK1. . . . . .                L NEAR  00FB    CODE
BLNK2. . . . . .                L NEAR  0103    CODE
CLER . . . . . .                N PROC  016E    CODE    Length =0009
CLRLP. . . . . .                L NEAR  0172    CODE
CONVRT . . . . . .              L NEAR  0000    CODE    External
CRLF . . . . . .                L BYTE  0023    CODE
DBLK . . . . . .                L BYTE  0019    CODE
DBUF . . . . . .                L BYTE  0003    CODE    Length =0007
DECMAL . . . . . .              L NEAR  0000    CODE    External
DLFIL. . . . . .                L NEAR  0170    CODE
DOBLNE . . . . . .              N PROC  013E    CODE    Length =000F
DOBRDR . . . . . .              N PROC  014D    CODE    Length =000F
DOLFTM . . . . . .              N PROC  015C    CODE    Length =0009
DOPRT. . . . . .                N PROC  00A5    CODE    Global  Length =0014
DORGTM . . . . . .              N PROC  0165    CODE    Length =0009
DPLP . . . . . .                L NEAR  00AB    CODE
FILL . . . . . .                L NEAR  0096    CODE
GONE . . . . . .                L NEAR  00EC    CODE
HIDF . . . . . .                L BYTE  000A    CODE
HIDNFL . . . . . .              V BYTE  0000    CODE    External
INLP . . . . . .                L NEAR  0069    CODE
LFTB . . . . . .                L BYTE  001C    CODE
NOFILL . . . . . .              L NEAR  009B    CODE
OTLP . . . . . .                L NEAR  0062    CODE
PELP . . . . . .                L NEAR  00C7    CODE
PNTR . . . . . .                V WORD  0000    CODE    External
PRHID. . . . . .                N PROC  0108    CODE    Length =0036
PRINT. . . . . .                N PROC  0027    CODE    Global  Length =007E
PRTEND . . . . . .              L NEAR  00B6    CODE
PRTENT . . . . . .              N PROC  00B9    CODE    Length =004F
PSX. . . . . .                  L BYTE  0000    CODE    Global
PUNT . . . . . .                L NEAR  00F8    CODE
RESTR. . . . . .                L BYTE  0001    CODE    Global
RGTB . . . . . .                L BYTE  0020    CODE
SETCNT . . . . . .              L NEAR  003E    CODE
STKCNT . . . . . .              V WORD  0000    CODE    External
TITLX. . . . . .                V BYTE  0000    CODE    External
TWOCH. . . . . .                L NEAR  0131    CODE

Warning Severe
Errors  Errors
0       0
```

End Listing Four

Listing Five

```
                                PAGE    ,132
                                TITLE   COENDP - Diskette contents list - Workarea Definitions
                                COMMENT                 * Version 1.0  -  June 1983
                                                                Modified - Feb 1984 *
                                PAGE    82
                        ;
0000                    CODE    SEGMENT PARA PUBLIC 'CODE'
                                ASSUME  CS:CODE, DS:CODE
                        ;
                                PUBLIC  SRCE, PNTR, RELOC
                        ;
= 0000                  RELOC   EQU     THIS BYTE               ; File extension reorder list
= 0029                  PNTR    EQU     RELOC+10*4+1            ; Pointer list
= 0119                  SRCE    EQU     PNTR+240                ; Start of entry stack
                        ;
0000                    CODE    ENDS
                        ;
                                END

Segments and groups:

                N a m e                 Size    align   combine class

CODE . . . . . .                        0000    PARA    PUBLIC  'CODE'

Symbols:

                N a m e                 Type    Value   Attr

PNTR . . . . . .                        E BYTE  0029    CODE    Global
RELOC. . . . . .                        E BYTE  0000    CODE    Global
SRCE . . . . . .                        E BYTE  0119    CODE    Global

Warning Severe
Errors  Errors
0       0
```

```
              IBM PC-DOS MASTER DISKETTE                Free:   7168   02/19/84

 BASIC.COM     11392  DISKCOPY.COM   2008  CALENDAR.BAS   3840  PIECHART.BAS   2304
 BASICA.COM    16768  EDLIN.COM      2392  CIRCLE.BAS     1664  SAMPLES.BAS    2432
 CHKDSK.COM     1720  FORMAT.COM     3816  COLORBAR.BAS   1536  SPACE.BAS      1920
 COMMAND.COM    4959  MODE.COM       2509  COMM.BAS       4352  EXE2BIN.EXE    1280
 COMP.COM       1649  SYS.COM         605  DONKEY.BAS     3584  LINK.EXE      41856
 DEBUG.COM      5999  ART.BAS        1920  MORTGAGE.BAS   6272
 DISKCOMP.COM   1640  BALL.BAS       2048  MUSIC.BAS      8704

                    Hidden files   2
```

End Listings

DR. DOBB'S CLINIC

by D.E. Cortesi, Resident Intern

Transcendental Sources

Some time back we published Richard Falk's request for sources of good algorithms for the transcendental functions. Several people responded with suggestions, notably R. C. Briggs, Thomas Chrapkiewicz, Tom Enterline, David Feign, Calvin L. Gardner, Glenn F. Roberts, and William A. Rutiser. Here are the titles they suggested that Falk read, with our comments on those we've used:

Approximations For Digital Computers, by Cecil Hastings, Jr., Princeton University Press, 1955. A very concise introduction to Chebyshev approximation methods, plus seventy-odd approximation formulae with graphs of their error functions.

BASIC Scientific Subroutines, Volumes I and II, by F. R. Ruckdeschel, McGraw-Hill, 1981.

Computer Approximations, by John F. Hart, et al., Robert E. Krieger Publishing Company, Krieger Drive, Malabar, FL 32950, 1968. Mentioned by two respondents. Rutiser sent a copy of the table of contents, which looks very complete.

Handbook of Mathematical Functions, edited by Milton Abramowitz and Irene A. Stegun, National Bureau of Standards Applied Mathematics Series #55, U.S. Government Printing Office, Washington, DC 20402.

Numbers In Theory and Practice, edited by Blaise W. Liffick, Byte Books. A rather miscellaneous collection of early *BYTE* articles, this is a useful introduction to floating-point arithmetic and to random number generation but not a complete or accurate source book.

Scientific Analysis on the Pocket Calculator, by Jon M. Smith, John Wiley and Sons, 1975. A friendly introduction to numerical analysis written for "those who owned or operated a modern electronic pocket or desk calculator" just as the age of the personal computer was dawning. Many approximation formulae with tables of error functions.

Software Manual for the Elementary Functions, by William J. Cody, Jr., and William Waite, Prentice-Hall, 1980. The book most often cited by our respondents and the one that comes closest to being a true cookbook, this is the reference of choice for Falk and anyone else who wants to write math functions in assembly language. Sixteen transcendental functions are presented as flowcharts for programs that can deal with float numbers at the bit or digit level; that is, at the level where the exponent and the fraction can be treated as separate integers. Where necessary, different algorithms are given for binary and decimal (BCD) methods. A great deal of attention is paid to accuracy. The function vocabulary is taken from Fortran, and some familiarity with that language is a help initially in getting into the book.

Over-Submitting

We're longtime advocates of using the batch facilities of one's operating system—SUBMIT and XSUB in CP/M 2.2, BAT files in MS-DOS, etc. Steve Grant of Hacienda Heights, CA, has the same bent, but he overloaded the XSUB facility of CP/M 2.2. Here's what he says:

"How do you do an XSUB file longer than 128 lines? Garden-variety SUBMIT files are easy since the last line of one file can just SUBMIT the next one. But that won't work with XSUB since the second SUBMIT command will be presented, not to the CCP, but to some innocent program that has never heard of the command line 'submit foo.'

"I got into this mess while trying to make my machine an Adventure Grandmaster. I carefully created a file as follows:

```
xsub
adv    (start the Adventure
       program)
...    (lots and lots of commands
...    to move around the cave,
       etc.)
```

I named it TROLL.SUB and did a SUBMIT TROLL.

"Disaster! It started executing well past the beginning without ever executing ADV. The CCP was thoroughly confused by the command GET SPICE. Upon closer examination it became clear the TROLL would always start executing 128 lines from the end. I threw out all but the first 128 lines. Now the program plays the first 126 moves just fine then stops and waits for human activity. Any ideas?"

Unfortunately, no. The problem lies in the peculiar format into which SUBMIT processes a submitted file and in the system's belief that a processed file won't exceed one logical extent of 16K (128 times 128). We could suggest that Steve convert to CP/M Plus or to MS-DOS version 2, both of which allow console input to be redirected to an ASCII file of any length, but we doubt that he'd find that constructive.

Or maybe he could find a way to make the Adventure program take multiple commands in one physical line? Reader suggestions are welcome . . . .

An Old Bug

Frequent correspondent David McLanahan got bit by an old bug. In fact, it's so old, and has been so faithfully transported to system after system, it might be useful to print his warning.

"I've created a number of .BAS files with lower-case names from MBASIC. They are a big bother; the directory lists them, but CP/M can't find, erase, or rename them. [*You can use BASIC's KILL command to delete them.*—DEC] A friend was over the other day and saved a program to disk in ASCII form. My friend meant to enter

SAVE "PROG",A

but by accident typed it as

SAVE "PROG,A

The final quote is optional—*when nothing follows the filename*! The file was created as "PROG,A.BAS" and not in ASCII form. From there it got into MAST.CAT. As you undoubtedly know, the catalog programs use commas as the delimiters between entries, thus the catalog got out of sync and blew 20K worth of listings! Mutter, mutter . . . ."

Do the newer Microsoft BASICs do this? Somebody check for us on a PC with DOS 2.1 and on a Macintosh, please.

[*Editor's Note: Upon reading this, we decided to try this ourselves on the IBM PC here in our office. Mr. McLanahan's problem has apparently been fixed on our BASICA for PC DOS 2.1. When we typed SAVE "PROG,A and hit return our request was refused and we received a somewhat cryptic "too many files" message. This, incidentally, is the same message we receive when asking BASICA to save a file with a filename containing over 10 characters plus the .BAS extension. Anyone know why this message is generated as opposed to one of the others that indicates a bad filename? — RW*]

BASIC Precision, Again

Several readers were bothered by our recent discussions of BASIC single- and double-precision numbers and the ills they are heir to. Bill Metzenthen, David S. Tilton, William R. Hamblen, and David Feign all wrote to make essentially the same point: namely, that an understanding of how binary float numbers are represented is essential to an understanding of how errors are formed and propagated. Let Tilton tell it, since he has a positive suggestion.

"What most people do not realize is that, when they talk about floating-point numbers in a computer, it is a binary point and not a decimal point that is floating around inside the machine.

"Everybody knows that there are some fractions that never come out even to any number of places. Thirds and sevenths, for example, cannot be expressed exactly as decimals. In the binary system even fewer fractions come out even. Usually these arise in money amounts where cents are expressed in hundredths of dollars. The only hundredths that will come out even as binary fractions are .00, .25, .50, and .75. All the others will have some small roundoff error no matter how many binary places (i.e., bits) are allotted to represent them.

"Fortunately there is a simple solution to the problem: store money amounts internally as an integral number of pennies. Not with the type integer, which would limit you to a maximum of $327.68. Single precision can devote 24 bits to an integer, and double precision can devote 56 bits. This is roughly equivalent to 7.2 and 16.8 decimal places, respectively. The only times you need to worry about roundoff errors are when you multiply by 100 on input and divide by 100 on output."

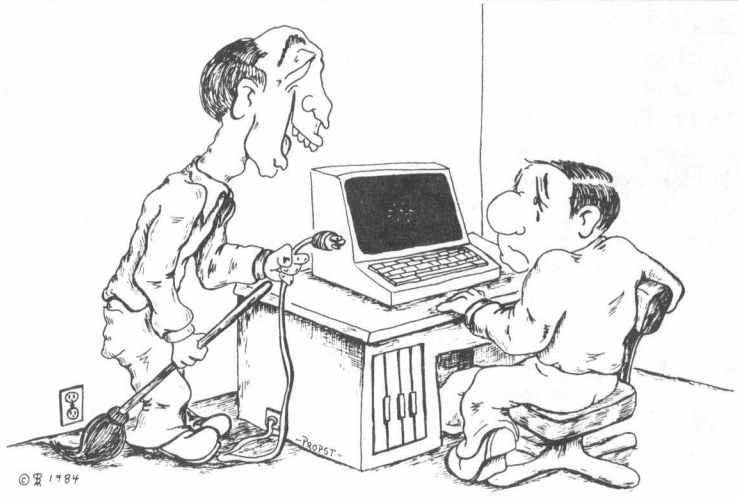
For the nit-pickers out there, we should qualify Tilton's statement about the fractions that can or can't be represented accurately. The fractions .00, .25, .50, and .75 are the only *two-digit* decimal ratios that can be represented exactly in binary. If you divide a quarter by two, the result is a complete and accurate representation of 0.125: twelve-and-a-half cents.

His solution (carry currency amounts as integral numbers of pennies) and his point that you can use floating-point variables to contain large integral values are both useful suggestions. There are pitfalls here as well, however. For example, if you multiply an integral number of pennies by our local sales tax rate of 0.065, you will have a fractional part in your result. The fraction represents not pennies but mills; nevertheless, it is subject to all the same problems of roundoff. Input can be a problem as well. Multiplying an input amount by 100 isn't sufficient; the input conversion routine might already have done some damage before you apply the scale factor.

Public Opinion Poll

Peter Baenziger of Kalamazoo, MI, takes us to task for the diatribe on polling loops we tacked onto the May column. Here's his counter-blast.

"You take Michael Barr to task for writing a polling loop checking the Caps Lock status on an IBM PC. You state, 'This is an example of how a solution can go wrong when it is implemented at the wrong level of the system.' You are probably right; it isn't the right level. But I don't think Barr and his loop are wrong. In my opinion, if the loop causes problems, it's a weakness in the multitasking operating



YOU KNOW, I NEARLY TRIPPED ON THIS SILLY THING!

system.

"Concurrent CP/M, for example, is sold for the IBM PC, and as such, supposedly supporting a specific computer, it should support the idiosyncracies, the strengths and weaknesses, of the hardware.

"We're living in an imperfect world. The imperfect computers are out there by the hundreds of thousands. An extremely small minority run Concurrent CP/M or Windows, etc. It's ludicrous, in my opinion, that a vast majority of users, millions of people, should always have to take a first peck at the keyboard to find out what the Caps Lock status is just so that a rare multitasking OS won't get into trouble.

"If the multitasking OS is properly designed, it will have saved the state of the machine, including the shift status, before relegating a task to the background. When the task moves to the foreground, the OS should restore the old shift status in 40:17, the polling loop will get it, and all will be well.

"Let me reiterate: I agree with you that two LEDs would be preferable. Or if not LEDs, then a proper DOS call. I even agree that some polling loops are troublesome in a multitasking environment. But the fact is, the two ideal solutions don't exist. Complaining that they should doesn't change reality. So I wouldn't call a program that brings a *solution* for *most* users a 'bad citizen.' Maybe what we need is better government."

Well, it's a thorny thicket of problems, and Baenziger has done us all a favor by pointing that out. There are lots of angles to explore. An important one, as Baenziger says, is the fact that all those PCs and PC clones exist, and most of them run (and will continue to run) plain old MS-DOS versions 1.2 or 2.0—single-task systems in which a polling loop won't cause any harm at all. Ditto for the near-million CP/M 2.2 systems and the near-million Apple DOSes.

Yet another angle is the culture from which the programmers come. On big systems, programmers tend to take the operating system interface as a given; they play by rules. People writing applications for an IBM 370 wouldn't dream of writing a polling loop. They know that their programs will be sharing the machine with many others; that the system as a whole will run more efficiently if their programs go to sleep as quickly as possible when they have nothing to do; that the less CPU time they consume, the higher their programs' priorities will become; and that every unnecessary machine cycle will be reflected in the user's timesharing bill.

None of these things are part of the gestalt knowledge associated with personal computers. Multitasking concepts just aren't in the vocabulary of the average grass-roots programmer.

Had IBM foreseen the coming of multitasking to the PC, they could have done something about that. For instance, they could have added a level of indirection to all those low-storage values (i.e., "For keyboard status, poll the byte whose *address* is at 40:17; to update the screen image, poke a byte *based on* the address at xyz."). That would have allowed an OS to set aside a machine-status block for each task, swapping the relevant pointers into low storage before activating a task.

The PC's success often makes us overlook the colossal marketing blunder that IBM almost made. If you look at the design, specs, and documentation of the PC circa 1982, you will realize that IBM intended the PC to be an up-market Apple. That's the framework for its BIOS design: one user, one program in BASIC (loaded from cassette), lots of peeks and pokes, fun and games, whee! Thanks largely to independent vendors, the machine turns out to have enough horsepower to be a lot more, but it is still saddled with that simple-minded internal design.

As a result, you get people like Yr. Intern who say, "Obviously a program should be written for good multitasking performance," and also people who say, "Obviously my program ought to use whatever it needs to make it work well."

Who's right? Both? Neither . . . ?

DDJ

SCISTAR: Greek and Math Symbols with WordStar

by John Thomas and Guy Fletcher

John Thomas, Brookhaven Instruments Corp., 200 13th Avenue, Ronkonkoma, NY 11779.
Guy Fletcher, The School of Mathematics & Physics, Macgarie University, North Ryde, NSW 2113, Australia.

We both bought new printers recently and were faced with learning a new set of user-programmed commands that would allow us to make use of some of the special features of these printers. Why not have a special set of commands that is universal—that is, the same for all printers—included with WordStar? Anyone who has used a text formatter such as SuperScript will recognize the value of such a command set. In such systems, a Greek alpha is embedded as \AL\; for Not Equal To, you just write \NE\; and so on. All the special codes for particular printers are transparent and invisible to the operator.

You can set up this same sort of command set very easily with WordStar, using the option MAILMERGE in a way probably never envisaged by MicroPro (yes, you can do more with WordStar than write hundreds of personalized letters to your relatives). At the same time that you are embedding these codes, you can include some of the special features of your printer, such as emphasized print or underscoring, in a much neater way than WordStar can. And the automatic word-wrapping still works perfectly, too. Furthermore, the use of universal and meaningful embedded commands greatly enhances the on-screen readability of scientific text.

Any Computer, Any Printer

One of us has a KayPro II with a ProWriter printer, the other a Morrow MD2 with a Mannesmann Tally 160L printer. You can use these special commands with any printer. Of course, there is a limited number (too limited!) of user-programmable functions in WordStar, and that number determines how many features you can fit in. First, there are the four user functions, USR1 to USR4, called by ^PQ, ^PW, ^PE, and ^PR. Then there are the ribbon-change toggle (two codes) on ^PY and the character-pitch change on ^PA and ^PN. And that's it: eight available code sequences—and one of those doesn't work from MAILMERGE (^PN sends a <CRLF>, for some reason that is obscure to us), though it works fine from WordStar. If each special feature needs one code sequence to turn it on and another to turn it off, then you can run only three special features, but that covers a lot of possibilities in alternate character sets.

The Set of Commands

The set of commands should be universal, so that you can take your disk to another computer and have it read and print your file. And you can sit down at a friend's computer without having to know what has been programmed on which keys or how to get Greek symbols. We tried to think of most of the common needs. Our complete set is listed in Table Ia and Ib (page 26 and 27).

Your printer doesn't offer some of these features? No matter, neither does ours. Include them anyway against the time when you have a better printer, and, meanwhile, print a blank (^PO) to leave space for hand-scripting them in. That way WordStar won't blow up when you print from someone else's disk. To embed any command in your text, just enclose the variable name in ampersands—for example, &al& for alpha. Because upper-case and lower-

SCISTAR Greek Symbols

	Lower Case		Upper Case	
Alpha	AL	α	BAL	A
Beta	BE	β	BBE	B
Gamma	GA	γ	BGA	Γ
Delta	DE	δ	BDE	Δ
Epsilon	EP	ϵ	BEP	E
Zeta	ZE	ζ	BZE	Z
Eta	ET	η	BET	H
Theta	TH	θ	BTH	Θ
Iota	IO	ι	BIO	I
Kappa	KA	κ	BKA	K
Lambda	LA	λ	BLA	Λ
Mu	MU	μ	BMU	M
Nu	NU	ν	BNU	N
Xi	XI	ξ	BXI	Ξ
Omicron	OC	o	BOC	O
Pi	PI	π	BPI	Π
Rho	RH	ρ	BRH	P
Sigma	SI	σ	BSI	Σ
Tau	TA	τ	BTA	T
Upsilon	UP	υ	BUP	Υ
Phi	PH	ϕ	BPH	Φ
Chi	CH	χ	BCH	X
Psi	PS	ψ	BPS	Ψ
Omega	OM	ω	BOM	Ω

Table Ia

case variables are treated the same, we chose &bal& ("big alpha"), etc., for the upper-case letters.

The Command File SCI.CMD

MAILMERGE requires two dot commands at the head of the document you want to merge-print: a data file instruction, .DF, to specify the file containing the actual codes, and a read variables instruction, .RV, to read the data file. We have about 100 different variables, which makes for several lines of typing, so we put them all in a command file. Insert this at the head of your text with ".FI SCI.CMD". This command file is the same for everyone and is given in Table II (page 27). There is nothing special about the order of the variables, but the data file *must* have the same order.

The Data File SCI.DAT

The file is individual to your printer. First, you have to decide what features you want and which patch areas to use for each feature.

For the Mannesmann Tally printer, we use ^PQ and ^PW to turn the emphasized (boldface) print option on and off. This is much better than using boldface from WordStar. We use ^PE and ^PR to turn the alternate (Greek) character set on and off, by setting and clearing bit 8 at the printer. We use the ribbon toggle ^PY to turn the in-line underscoring option on and off; again, this works much better than underscoring from WordStar. And we use ^PA and ^PN to disable and enable the paper-out switch, for printing single sheets. We have to use these two controls from WordStar directly, as they are not part of the MAILMERGE file—they wouldn't work from there, anyway!

Likewise, for the ProWriter we prefer to use its own boldfacing capability via ^PQ (on) and ^PW (off). ^PE turns on the Greek and ^PR turns it off (actually, it turns the ASCII character set back on). Again, we use ^PY to toggle the ProWriter's own in-line underscoring on and off. At the moment, we aren't using ^PA and ^PN for anything useful, but we could use them (outside of MAILMERGE) to enable elite characters, for example.

To write the data file, it helps to understand how WordStar processes the hex codes that are stored in your text file or embedded through MAILMERGE. All the printable characters from 20H to 7EH are sent directly to the printer. The 26 codes from 01H to 1AH are the 26 control codes ^A through ^Z; WordStar operates directly on these. Thus, 13H (^S, underscore) sends a backspace 08H followed by the underscore character 5FH after every character until you turn off underscoring. ^Q, ^W, ^E, ^R, ^Y, and ^A (but not ^N if you are in MAILMERGE) send the code sequences that you have patched in WordStar to the printer.

The files SCI.CMD and SCI.DAT are most conveniently written under WordStar in nondocument (N) mode. Put a hard <CRLF> at the end of every line; this is especially important to include with the last line in each file. To write ^Q into your data file, you must type ^PQ so that ^Q is actually printed; otherwise, WordStar will act on ^Q directly instead of printing it to the data file.

The two tables in the appendix (page 29) show the actual data files SCI.DAT used for each of our printers.

SCISTAR Math Symbols

Multiply	X	×	Left Arrow	L	←
Divide	DI	÷	Right Arrow	R	→
Plus/Minus	PM	±	Up Arrow	U	↑
Not Equal	NE	≠	Down Arrow	D	↓
Identical	ID	≡	Dagger	DA	†
Approx. Equal	AE	≅	Half	HA	½
Greater/Equal	GE	≥	Quarter	QU	¼
Greater/Order	GO	≳			
Less/Equal	LE	≤	Up 0	U0	0
Less/Order	LO	≲	Up 1	U1	1
Approx.	AP	≈	Up 2	U2	2
			Up 3	U3	3
Infinity	IF	∞	Up 4	U4	4
Proportional	PR	∝	Up 5	U5	5
Centre Dot	CD	·	Up 6	U6	6
Convolution	CV	⊕	Up 7	U7	7
Degrees	DG	°	Up 8	U8	8
Square	SQ	■	Up 9	U9	9
Grad	GR	▽	Up n	UN	n
Partial Deriv.	PD	∂	Up left brckt	UL	(
Square Root	RT	√	Up right brckt	UR	)
Therefore	TF	∴	Up Plus	UPL	+
Integral	INT	∫	Up Minus	UM	-
			Up Dot	UD	.
Bold On	BON		Up Asterisk	UA	*
Bold Off	BOF		Up Slash	US	/
Underline	UND	(toggle)			

Table Ib

Command File SCI.CMD

```
.DF SCI.DAT
.RV AL,BE,GA,DE,EP,ZE,ET,TH,IO,KA,LA,MU
.RV NU,XI,OC,PI,RH,SI,TA,UP,PH,CH,PS,OM
.RV BAL,BBE,BGA,BDE,BEP,BZE,BET,BTH,BIO,BKA,BLA,BMU
.RV BNU,BXI,BOC,BPI,BRH,BSI,BTA,BUP,BPH,BCH,BPS,BOM
.RV BON,BOF,UND
.RV X,DI,PM,NE,ID,AE,GE,GO,LE,LO,AP
.RV IF,PR,CD,CV,DG,SQ,GR,PD,RT,TF,INT
.RV L,R,U,D,DA,HA,QU
.RV U0,U1,U2,U3,U4,U5,U6,U7,U8,U9,Un
.RV UL,UR,UPL,UM,UD,UA,US
```

Table II

If you have a different printer, you will have to discover your own code sequences to use and, of course, patch them into WordStar. In the appendix we also show how to patch WordStar to configure your printer in the same way as ours.

The Problems

You may have a printer different from either of the ones we've discussed, but you can still learn from our early mistakes with each printer. At the very least, you will become acquainted with the macabre way in which the eccentricities of WordStar and your printer may interact.

Mannesmann Tally 160L

Invoke the D-codes (daisy) option on the printer and patch ROLUP and ROLDOW in WordStar to ESC D and ESC U, respectively (yes, that *is* right). Then you can use the normal superscript and subscript toggles in WordStar. You *must* install WordStar if you have a teletype-like printer that can backspace; otherwise, you will not be able to overstrike or backspace in any line containing a subscript because of the way the printer handles superscripts and subscripts. You cannot backspace inside a superscript or subscript either; this fact unfortunately prevents superscripting composite characters such as Φ (overprint o and |).

Underscoring with &und& is done in-line by the printer—without that repeated backspacing that shakes the whole table—and boldfacing with &bon& and &bof& is done by a smooth extra pass over the line to boldface the required words. Don't try to avoid repeated backspacing by installing WordStar for a nonbackspacing teletype-like printer, because WordStar will then send an overprint line *after* the printer has rolled down to the subscript line!

When printing a line containing superscripts and subscripts, the Mannesmann Tally 160L printer first rolls down a full line, then prints the superscripted part of the line, rolls down a quarter of a line, prints the main line, rolls down another quarter of a line, and finally prints the subscripted part of the line before rolling down a full space to the next line. So for every line containing superscripts or subscripts, the printer rolls down an extra quarter-line: all of this is unknown to WordStar. This may well take you past the perforation at the end of a page, unless you shorten the page length from 66 to, say, 62 lines. Also, you *must* use the form-feed option when printing, to position the top of the next page correctly.

There is a fundamental problem of incompatibility between backspacing and superscripting or subscripting in the same line. You just can't superscript around or after a backspace or subscript before or around one. Those few backspaces hidden in SCI.DAT for making composite characters by overstriking are included in this conundrum. The only way a round this seems to be to overprint a complete new line from WordStar (^P RETURN) that contains the desired superscripts and subscripts.

ProWriter

The ProWriter needs to be installed carefully, though not especially because of our scheme. It has to do mainly with the way in which the ProWriter prints its Greek characters. Basically, you cannot have multiple passes over any whole line of text that has a Greek character in it, because when WordStar makes multiple passes (e.g., to boldface), it inserts spaces (20H) in any position that is not being overprinted. This normally is no problem, but when Greek is enabled, the ProWriter prints ∝ when it receives 20H because WordStar always remembers which character set was enabled at each and every point in the line.

To circumvent these problems, you must install the ProWriter as a teletype-like printer that can backspace, and you must specify that superscripting and subscripting be done via WordStar's ROLUP and ROLDOW. This means that the printer *must* be initialized (via PSINIT) to enter the incremental print mode, and you need to set up ROLUP and ROLDOW to move a half-line up and down (details in the appendix).

MAILMERGE will not pass ″ (22H) as a variable, so to get Greek γ from the ProWriter, you have to do it longhand via WordStar (i.e., ^PE″^PR).

By the way, the ProWriter's own boldfacing and underscoring work beautifully in a single line pass, but rolling up and down is a flaky operation on our ProWriter. The printer rolls up erratically; often, the vertical registration can be out by a quarter of a character (big minus for ProWriter).

It Works

We have already written a couple of scientific papers using this scheme. It works beautifully, producing readable source files and the desired output. For example,

```
x^T&al&^T &ge& &be&&u2&
is an &bon&inequality&bof&
```

produces this line:

$x^{\alpha} \geq \beta^2$
is an **inequality**.

Nice, huh? ■■J

APPENDIX

Data File SCI.DAT for Mannesmann Tally 160L

```
^E^R,^Ea^R,^O,^Ek^R,^En^R,^O,^O,^Ei^R,i,^O,^O,^Ef^R
^O,^O,o,^Ec^R,^O,^Ee^R,^Eg^R,^O,o^H¦,^O,^O,^O
A,B,^Eb^R,\^HL,E,Z,H,^O,I,K,^O,M
N,^O,O,^Eo^R,P,^Ed^R,T,Y,^Eh^R,X,^O,^Ej^R
^Q,^W,^Y
X,^Ev^R,^Eq^R,=^H/,^Ep^R,^O,^Er^R,>^H^V^V,^Es^R,<^H^V^V,^Ew^R
^El^R,^O,^E^R,O^H+,^Ex^R,^O,^T-^T^HV,o^H^E{^R,^O,^E^Tt^T^H^Vu^V^R
^O,^O,^O,^O,-^H¦,1/2,1/4
^Ex^R,^T1^T,^E}^R,^T3^T,^T4^T,^T5^T,^T6^T,^T7^T,^T8^T,^T9^T,^E^R
^T(^T,^T)^T,^T+^T,^T-^T,^T.^T,^T*^T,^T/^T
```

Wordstar Patches for Mannesmann Tally 160L

Bold_on	USR1	ESC[=z	04H,1BH,5BH,3DH,7AH
Bold_off	USR2	ESC[>z	04H,1BH,5BH,3EH,7AH
Greek_on	USR3	ESC[9z	04H,1BH,5BH,39H,7AH
Greek_off	USR4	ESC[8z	04H,1BH,5BH,38H,7AH
Paper_out_disable	PALT	ESC[:z	04H,1BH,5BH,3AH,7AH
Paper_out_enable	PSTD	ESC[<z	04H,1BH,5BH,3CH,7AH
Underscore_on	RIBBON	ESC[4m	04H,1BH,5BH,34H,6DH
Underscore_off	RIBOFF	ESC[0m	04H,1BH,5BH,30H,6DH
Superscripting	ROLUP	ESCD	02H,1BH,44H
& subscripting	ROLDOW	ESCU	02H,1BH,55H

Printer installed as "teletype-like printer that can backspace"
D-codes enabled on printer

Data File SCI.DAT for Prowriter

```
^E^R,^E!^R,^O,^E#^R,^E$^R,^E%^R,^E&^R,^E'^R,^E(^R,^E)^R,^E*^R,^E+^R
^E",^R,^E-^R,^E.^R,^E/^R,EO^R,^E1^R,^E2^R,^E3^R,^E4^R,^E5^R,^E6^R,^E7^R
A,B,^E9^R,^E8^R,E,Z,H,^O,I,K,^E;^R,M
N,^O,O,^O,P,^E:^R,T,Y,^O,X,^P,^E<^R
^Q,^W,^Y
X,^O,^ED^R,^EE^R,^O,^O,^EF^R,>^H^V^V,^EG^R,<^H^V^V,^EH^R
^EK^R,^O,^E1^R,^EJ^R,^EO^R,^E?^R,V^H^T-^T,o^H,^E>^R,^EL^R,^O
^EB^R,^EC^R,^E@^R,^EA^R,^H-,^EM^R,^EN^R
^EO^R,^EP^R,^EQ^R,^ER^R,^ES^R,^ET^R,^EU^R,^EV^R,^EW^R,^EX^R,^Tn^T
^EY^R,^EZ^R,^E[^R,^E\^R,^E]^R,^E^^R,^E_^R
```

WORDSTAR PATCHES for ProWriter

Bold_on	USR1	ESC!	02H,1BH,21H
Bold_off	USR2	ESC"	02H,1BH,22H
Greek_on	USR3	ESC&	02H,1BH,26H
ASCII on	USR4	ESC$	02H,1BH,24H
	PALT		
	PSTD		
Underscore_on	RIBBON	ESCX	02H,1BH,58H
Underscore_off	RIBOFF	ESCY	02H,1BH,59H
Superscripting	ROLUP	ESCr<LF>	04H,1BH,72H,0AH,00H
& subscripting	ROLDOW	ESCf<LF>	04H,1BH,66H,0AH,00H
Normal <CRLF>	PSCRLF	ESCf<CR><LF><LF>	05H,1BH,66H ODH,OAH,OAH
Initialization string	PSINIT	ESC[ESCT12	07H,0DH,1BH,5BH 1BH,54H,31H,32H
Finish string	PSFINI	ESC]	02H,1BH,5DH

Printer installed as "teletype-like printer that can backspace"
PSINIT sets ProWriter in incremental mode and sets linefeed to 1/12".
PSFINI returns ProWriter to logic seek print mode.
PSCRLF gives a "normal" <CRLF> by making sure that the carriage rolls down, and sending two 1/12" linefeeds.

What's the Diff?

A File Comparator for CP/M Plus

A file comparator is a program that compares two files and reports the differences between them, or reports that they have no differences. Such a program can be very useful, especially in@s×stems that don't put a time stamp on each file that is created. In such systems a comparator is often the only way to answer such questions as, "Is that *really* the latest version?".

A file difference finder is a program that goes one step further. Not only does it find the differences between two files, it records those differences in a third file for later use. A difference finder is a key part of any system for the automatic management of software development. A program source file goes through many changes as it is developed, tested, and maintained. With a difference finder, each successive version of a program can be stored as a difference file that contains only the changes from the prior version. There's great storage economy in this. Furthermore, a difference file is a compact representation of a program update, and as such it can be a good way of distributing a bug-fix.

The ideal format for a difference file is a script of commands that will direct some kind of editor in the process of converting the old file into the new one. When differences are captured in such a form, the latest version of a base file can be created entirely by software, automatically. The editor is applied to the base file and given successive difference files as its script. The edited result is the new file.

For example, in IBM's VM/370 operating system difference files are called "update" files, and there is a special UPDATE program that will generate a new file from a base file and a list of update files. Text editors used with the system have the ability to store only the changes made during an edit session, writing them as an update file.

A system like the one in VM/370 requires special-purpose software (the UPDATE program is used only for that, and update files are not the normal output of an edit). A public domain program called DIFF can be found on some bulletin boards; it, too, seems to use a special-purpose program to apply the updates. How much nicer it would be if a file difference system could be constructed using only the standard tools a system affords! No doubt this can be done in Unix, with its plethora of tool programs and its great ease of plugging tools together. But it can also be done in CP/M Plus.

CP/M Plus contains most of the tools that are needed. It has ED, an editor that nobody would willingly adopt for personal use but one that is perfectly suitable as an automatically driven file updater. Furthermore, CP/M Plus contains the command GET, which redirects console input to a file. Thus these two CP/M Plus commands:

```
get differ.dat
ed basefile
```

would cause ED to load BASEFILE and perform upon it the modifications commanded by the lines of DIFFER .DAT. If there were several difference files to be applied in order, they could be handled by a sequence of commands, and the sequence could be written up as a submit file to run automatically. All that is lacking is an automatic way to create difference files.

What is needed is a program that will read an old and a new file (most likely the new file will be the result of editing the old one, and the old file will be the .BAK version of the new one). After comparing the two files, the program will write a difference file; this will contain exactly the ED command lines needed to transform the old file into the new one.

It sounds fairly straightforward, doesn't it? Before you read any further, take a few minutes to sketch out the top-level logic of such a program. I'll wait.

Ah, you're back. Not so simple after all, is it? As a matter of fact, it's a bear. The intuitive algorithms (and some that are not so intuitive) all seem to lead to excessive storage use, or to execution times that are an exponential function of the file sizes. One common method uses a comparison "window" whose size is a critical program parameter. Here's a quote from the manual for AUTODIFF, a file comparator distributed by The Software Toolworks.

"AUTODIFF starts to work by lining up the tokens from one input file . . . with the tokens from the other input file. Next it looks for the start of a difference . . . . The next task is the hard part: finding where the difference ends. This is called synchronizing the files, and AUTODIFF's tool for accomplishing it is the sync window . . . . Starting at the head of each queue, AUTODIFF slides a sync window back and forth over each queue until it finds the same pattern showing through both windows, or until one or both of the windows slide off the end of its queue.

"If the sync windows were very small . . . AUTODIFF would most likely see the same pattern in both windows even though the surrounding tokens were totally different. On the other hand, if the sync windows were very large . . . AUTODIFF would most likely slide both windows right off the ends of their queues, in which case its idea of a difference would consist of the whole of each input file. So the size

by D.E. Cortesi

D.E. Cortesi, 430 Sherman #212, Palo Alto, CA 94306.

of the sync window is very important. For AUTODIFF to work, it can't be too large or too small . . . . Unfortunately, the right sync window size depends heavily on the statistics of the data . . . ." (Copyright 1982, Digital Constructs, Inc.)

After making a number of stabs at the problem, I decided to turn to "the literature" to find out if somebody smarter than I had found a better answer. And somebody had. The somebody is one Paul Heckel, who published his algorithm in a paper entitled "A Technique for Isolating Differences Between Files" in the April 1978 issue of *Communications of the ACM*. Heckel's algorithm actually generates *more* data than a difference finder needs. Not only that, but it does it in time that is guaranteed to be linear in the size of the files and with storage that is never more (usually much less) than the combined file sizes.

Heckel started by reversing the terms of the problem. Rather than searching for *differences* between two files, his algorithm concentrates on finding the *similarities*. Once those are marked, whatever is left is a difference. He devised two rules that would find most similarities.

Rule 1 says that "a line that appears once and only once in each file must be the same line (unchanged but possibly moved)." The rule is implemented using a symbol table and two arrays.

First, the old file is read and each of its lines is entered into the symbol table. An array called *OA*, indexed by the line numbers of the file, is used to record the symbol-table numbers of each line. Only the text of unique lines needs to be retained; duplicated lines simply get duplicate symbol-table indices. A counter in each symbol-table entry is incremented each time a line is found.

Next the new file is read and its lines are entered to the symbol table. Only line texts unique to the new file need to be retained. Another counter in each symbol-table entry is incremented to record the appearance of each new-file line, and the file is recorded as an array *NA* of symbol-table indices.

Now rule 1 can be applied. For each line represented by *NA*, look at the corresponding symbol-table entry. If the old and new counters both contain 1, that line satisfies rule 1 (appears once and only once in each file). The corresponding new and old lines are marked as "matched"; they cannot be differences. Lines are "matched" by putting reciprocal line numbers in their *NA* and *OA* array entries.

At this point, four kinds of lines are known. Those that satisfy rule 1 are marked. The remainder either appear in the old file but not the new (and hence are deletions); or appear in the new file but not the old (these are insertions); or appear more than once in one or both files (and are ambiguous at this point). To understand this, think of the block comments in a program file. The comment text lines, if they are unchanged, would be matched up by rule 1. The decorative lines of comment delimiters that frame them would not be matched by rule 1 because many of them appear in each file.

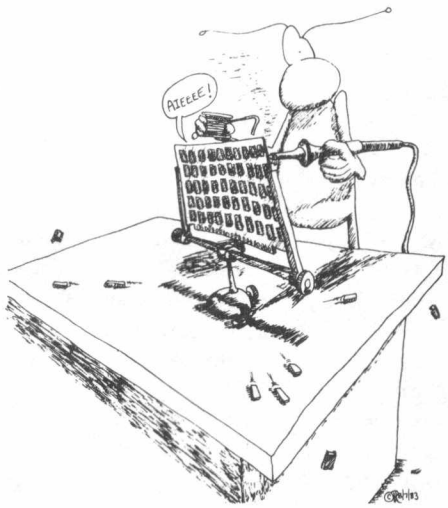
Heckel's rule 2 acts to resolve the last category. I would phrase it this way: "If the lines adjacent to a matched pair of lines are identical, then the adjacent pair are matching lines as well." These adjacent matches can be found in two sweeps over the arrays, one to find matches that follow a matched line and the other to find those that precede one. Afterward, all the similar lines have probably been found. The only similarities that would be missed are those that (a) appear more than once in each file and (b) are separated from every rule-1 line by a changed line. Such missed similarities will not be lost, but they will be treated as changed lines in the output.

Heckel claims as an advantage of his algorithm that it recognizes block moves for what they are. Lines that are moved, but not otherwise changed, are matched as similar. They can be recognized by discontinuities in the ascending sequence of line numbers in both the *OA* and *NA* arrays. Unfortunately, there is no easy way to record a block move as a sequence of ED commands, so I couldn't use the information. However, it is possible to locate all block moves and "unmatch" them (converting them into deletes and inserts) in a single sweep over the arrays.

The listing that follows (page 32) is a prototype implementation of Heckel's algorithm in Pascal. Since it was put together as a learning exercise, it has a number of rough edges; still, it does work. Except for a few compiler dependencies (declaring a variable-length string, finding its length, and opening files) it should work with any CP/M or MSDOS Pascal compiler.

Pascal is not the best language for this program. It really needs more freedom to allocate storage. An implementation in C could be more flexible about maximum file sizes. An assembly language version could play several tricks to reduce the size of the symbol-table entries as well. Even so, this version can do useful work on files of moderate size, and it is surprisingly quick in execution, running in only a little more time than it takes to read the two files and write the difference file. ■■J

(Listings begin on next page)



What's the Diff? Listing (Text begins on page 30)

```
{       HDIFF : a file-difference finder

    Based on "A Technique for Isolating Differences
    Between Files" by Paul Heckel, Communications of
    the ACM Vol 21 Number 4 (April 1978), this program
    (1) reads an old and a new file, (2) finds the diff-
    erences between them in time that is linear in the
    size of the files, and (3) writes a file which is
    a script to drive an editor (ED) to change the old
    into the new.

}
program diff(input,output);

label 99;   { main-proc error exit }

const
    maxfile = 300;       { largest old/new file (arbitrary) }
    maxover = maxfile+1;{ allow for a sentinel value }
    maxchar = 127;       { max chars/line, 128 bytes/string }
    maxsym = 1023;       { symbol table size, prime > 2*maxfile }
    topsym = maxsym-1;   { max index of ST array }

{ CP/M storage requirements are: 6*maxover + 8*maxsym plus
  128 bytes per each UNIQUE line in the combined files }

type
    symnum = 0..topsym;      { type of symbol table indices }
    linenum = 1..maxover;    { types of indices to OA, NA }
    linecnt = 0..maxover;

{ compiler dependent: define a varying string up to maxchar bytes }
    ltext = string maxchar;

    linerec = record     { descriptor of one file line in OA, NA }
        matched : boolean;  { true when matched with other file }
        index : integer;    { index to symtab or to other file }
    end;

    symrec = record      { one symbol table entry }
        hashval : integer;  { hash of this line, neg = unused entry }
        lineval : ^ltext;   { address of text string }
        Oline   : linenum;  { index to line in old file OA }
        Ocount  : 0..2;     { occurrences in old file: 0, 1, many }
        Ncount  : 0..2;     { ditto in new file }
    end;

var
    oldmax, newmax : linecnt;   { linecount of each file }
    oldfile, newfile, diffile: text;
    OA, NA : array[linenum] of linerec;
    ST : array[symnum] of symrec;

{ Compute a 15-bit signature based on all the characters of a
```

(Continued on page 34)

What's the Diff? Listing (Listing continued, text begins on page 30)

```
  line.  Pick a symbol table entry based on the hash and return
  the symbol index.  If different lines hash to the same entry
  just overlow by ones, the simplest policy.  The full hash
  code is saved in the symtab entry for quick discovery of
  unequal lines. }

function store(var t: ltext) : symnum;
    label 1,2,3;
    var s: symnum; h: integer;

    { compiler-dependent: return length of string t }
    function strlen(var t: ltext): integer;
        type pascalz = string 255;
        function length(pz: pascalz):integer; external;
        begin strlen := length(t) end;

    function hash(var t: ltext): integer;
        const hashlim = 16256; { (maxint div 4)-127 }
        var q: integer; len: 0..maxchar;
        begin
            q := 0;
            for len := strlen(t) downto 1 do begin
                q := q + ord(t[len]); { add current byte }
                { shift left to 14 bits, then wrap }
                if (q < hashlim) then q := q + q
                                 else q := 1 + q div 2
            end;
            hash := q
        end; { function hash }

    begin { function store }

        h := hash(t);
        s := h mod maxsym;  { initial probe of symbol table }

    { find duplicate line or vacant symbol starting at ST[s].
      If you know a "structured" way to code this, let me know. }

    1:  if (ST[s].hashval < 0 ) then goto 2; { free entry }

        { used entry, is t a duplicate of another line? }

        if (ST[s].hashval = h) { quick test, eliminates most }
        and(ST[s].lineval^ = t) { slow test assures dup line }
            then goto 3; {yes, we have this line}
        s := (s+1) mod maxsym; { try next table entry }
        goto 1;

    2: { empty symbol table entry found -- install new line }
        with ST[s] do begin
            hashval := h;
            new(lineval);
            lineval^ := t
        end;
```

```
3: { line now exists in symbol table }
    store := s
    end;

{ read the old file, store its lines in the symbol table, and
  count their occurrences in Ocount in each entry. }
procedure pass1;

    var o: linecnt;
        s: symnum;
        t: ltext;

    begin
        o := 0;
        repeat
            readln(oldfile,t);
            o := o+1;
            s := store(t);
            with ST[s] do begin
                Oline := o;
                if (Ocount <> 2) then Ocount := Ocount + 1
            end;
            with OA[o] do begin
                matched := false;
                index := s
            end
        until ( eof(oldfile) or ( o = maxfile ) );
        with OA[o+1] do begin { create sentinel }
            matched := true;
            index := maxover
        end;
        oldmax := o

    end;

{ read the new file, store its lines in the symbol table, and
  count their occurrences in Ncount in each entry. }
procedure pass2;

    var n: linecnt;
        s: symnum;
        t: ltext;

    begin
        n := 0;
        repeat
            readln(newfile,t);
            n := n+1;
            s := store(t);
            with ST[s] do
                if (Ncount <> 2) then Ncount := Ncount + 1;
            with NA[n] do begin
            matched := false;
            index := s
        end
    until ( eof(newfile) or ( n = maxfile ) );
    with NA[n+1] do begin { create sentinel }
        matched := true;
        index := maxover
```

(Continued on next page)

```
        end;
        newmax := n

    end;

{ record the fact that two lines match by placing
  reciprocal indices in their array entries. }

procedure matchup(o, n: linenum);
    begin
        with OA[o] do begin
            matched := true;
            index := n
        end;
        with NA[n] do begin
            matched := true;
            index := o
        end
    end;

{ apply rule 1: when a line appears just once in each file, it is
  the same line (altho perhaps not in its original position).}
procedure pass3;

    var s: symnum; o,n : linenum;

    begin
        for n := 1 to newmax do begin
            s := NA[n].index;
            with ST[s] do
                if (Ocount=1) and (Ncount=1) then
                    matchup(Oline,n)
        end
    end;

{ apply rule 2, sweeping down the file to spread the rule-1 matches
  toward the end of the file. }
procedure pass4a;

    var o, o1, n, n1 : linenum;

    begin
        for n := 1 to newmax-1 do
            if (NA[n].matched) then begin
                n1 := n + 1;
                if (not NA[n1].matched) then begin
                    o := NA[n].index;
                if (o < oldmax) then begin
                    o1 := o + 1;
                    { the "matched" flags participate in
                      the following comparison to ensure
                      that symbol indices are compared }
```

```
                        if (OA[o1] = NA[n1]) then
                            matchup(o1,n1)
                    end
                end
            end
    end;

{ apply rule 2, sweeping up the file to spread previous matches
  toward the top of the file. }
procedure pass4b;

    var o, o1, n, n1 : linenum;

    begin
        for n := newmax downto 2 do
            if (NA[n].matched) then begin
                n1 := n - 1;
                if (not NA[n1].matched) then begin
                    o := NA[n].index;
                    if (o > 1) then begin
                        o1 := o - 1;
                        if (OA[o1] = NA[n1]) then
                            matchup(o1,n1)
                    end
                end
            end
    end;

{ Unmatched lines in OA represent deletes; unmatched NA lines are
  inserts.  Ignoring these, the matched lines should increase
  monotonically.  When they don't, when a discontinuity appears,
  a block-move is present.  It is unclear how to record such
  moves in a difference file, so here we find them and "unmatch"
  the moved block, converting it into a delete/insert.

  There is a figure/ground ambiguity -- any block move can be
  seen as a move-up of some lines or a move-down of others.  We
  select the smaller of the two groups to be the "moved" group.}

procedure pass5;

    var o, n: linenum;

    procedure resolve( var o, n : linenum );
    { a discontinuity starts at OA[o] and NA[n]. figure out whether
      fewer lines have been moved up to NA[n] or down from OA[o],
      and convert the smaller group to inserts and deletes. }

        var xo, xn : linenum;

            t, ospan, nspan : integer;
            s : symnum;
```

(Continued on next page)

What's the Diff? Listing (Listing continued, text begins on page 30)

```
    begin
        xo := o;    { measure the block starting at OA[o] }
        repeat
            t := 1 + OA[xo].index;
            xo := xo + 1
        until (t <> OA[xo].index);
        ospan := xo - o;
        xn := n;    { measure the block moved up to NA[n] }
        repeat
            t := 1 + NA[xn].index;
            xn := xn + 1
        until (t <> NA[xn].index);
        nspan := xn - n;
        if (ospan < nspan) then begin   { a block moved down }
            xo := o;            { convert OA[o..] which matches }
            xn := OA[o].index;  { lines further down in NA }
            t  := ospan;        { number of lines to convert }
            o  := o + ospan     { continue scan after block }
        end
        else begin                      { a block moved up }
            xn := n;            { convert NA[n..] which matches }
            xo := NA[n].index;  { lines further down in OA }
            t  := nspan;        { number of lines to convert }
            n  := n + nspan     { restart scan after block }
        end;
        { Unmatch a block of matched lines.  We need the
          symbol table index here, and find it by scanning
          the table.  This expensive operation could be
          eliminated by retaining the symbol index in OA}

        while ( t > 0 ) do begin
            s := 0;
            while (ST[s].Oline <> xo) do s := s + 1;
            with OA[xo] do begin
                matched := false;
                index := s
            end;
            with NA[xn] do begin
                matched := false;
                index := s
            end;
            xo := xo + 1;
            xn := xn + 1;
            t := t - 1
        end
    end; { procedure resolve }

begin { procedure pass5 }
    o := 1; n := 1;
    while (OA[o].index <> maxover) do begin

        while (not OA[o].matched) do {skip deletes}
            o := o + 1;
        while (not NA[n].matched) do {skip inserts}
            n := n + 1;
```

```
            if (OA[o].index = n) then begin { matching lines }
                o := o + 1;
                n := n + 1
            end
            else { a discontinuity ? }
                if (OA[o].index <> maxover) then { yes }
                    resolve(o,n)
                else { no, just the sentinel }
        end

    end; { pass5 }

{ Write the difference file as a script that will drive a
  line-editor in converting the old file to the new one.
  Actual output is modularized for ease in converting to
  a different editor than CP/M's ED. }

procedure pass6;

    var o, n, xn : linenum;
        t : integer;

    { cause necessary initialization of editor }
    procedure putfirst;
        begin
            writeln(diffile,'#a');
        end;

    { delete t lines starting at current one, leave
      line after last delete as current line }
    procedure putdel(t : integer);
        begin
            writeln(diffile,t:1,'k')
        end;

    { set up to insert lines at end of file }
    procedure putbot;
        begin
            writeln(diffile,'-b')
        end;

    { skip ahead t lines in the file }
    procedure putmov(t : integer);
        begin
            writeln(diffile,t:1,'L')
        end;

    { insert t lines beginning with NA[k] -- the new
      lines PRECEDE the current line and afterward the
      current line is the same as at the start. }

    procedure putins(k : linenum; t : integer);
        begin
            while (t > 0) do begin
                writeln(diffile,'i',ST[NA[k].index].lineval^);
                k := k + 1;
```

(Continued on page 42)

```
            t := t - 1
        end;
    end;

{ close file and end editor }
procedure putlast;
    begin
        writeln(diffile,'e');
    end;

begin {pass6}
    putfirst;
    o := 1; n := 1;
    repeat

        if (not OA[o].matched) then begin { deleting }
            t := 0;
            repeat
                t := t + 1;
                o := o + 1
            until (OA[o].matched);
            putdel(t)
        end;

        if (not NA[n].matched) then begin { inserting }

          { if (n = 1) then puttop? -- nothing special
            needed to insert above line 1 with ED }

            if (o > oldmax) then { ins. at bottom }
                putbot;

            t := 0;
            xn := n;
            repeat
                t := t + 1;
                n := n + 1
            until (NA[n].matched);
            putins(xn,t)
        end;

        t := 0;
        while (OA[o].matched) and
              (NA[n].matched) and
              ( o <= oldmax ) do begin { skipping unchanged lines }
            t := t + 1;
            o := o + 1;
            n := n + 1
        end;
        if (t>0) then putmov(t)

    until (OA[o].index=maxover) and (NA[n].index=maxover);
    putlast;
end;

procedure setup;
```

```
    var j : symnum;
        fspec : string 14;

    begin

        for j := 0 to topsym do
            with ST[j] do begin
                hashval := -1; { entry not in use }
                Oline := maxover;
                Ocount := 0;
                Ncount := 0
            end;

        oldmax := 0;
        newmax := 0;
{ compiler-dependent: get names and open files }
        write('ORIGINAL file''s spec: '); readln(fspec);
        reset(fspec, oldfile);
        write('MODIFIED file''s spec: '); readln(fspec);
        reset(fspec, newfile);
        write('DIFF file''s spec: '); readln(fspec);
        rewrite(fspec, diffile)
    end;
begin {main procedure: set up, check for empty files, invoke passes }

    setup;
    if (eof(oldfile)) then begin
        writeln('Old file is empty; diff is all-insert.');
        goto 99
    end;
    if (eof(newfile)) then begin
        writeln('New file is empty; diff is all-delete.');
        goto 99
    end;
    pass1;  { read, store old file }
    pass2;  { read, store new file }
    pass3;  { apply rule 1 }
    pass4a; { apply rule 2 working down }
    pass4b; { apply rule 2 working up }
    pass5;  { convert moves to del/ins pairs }
    pass6;  { generate script for ED }
99:
end.
```

End Listing

An Infinite Key Encryption System

If you have ever stored private information on a public disk file or transmitted it over the telephone network to storage on a mainframe or to a correspondent, you may have wondered if your private information would remain private. Who else is reading it? Your co-workers? Other subscribers? The system programmers? The government?

Trusting souls who have never felt the slightest concern about such matters may read no further; others may be interested to learn how to protect themselves against such intrusion and theft. This article discusses the construction of a strong file encryption system for CP/M, which we call "TNT."

Several encryption programs for microcomputers are now on the market. To our knowledge, however, none of their authors has made public the method used or has offered any argument to support the claim that the system is indeed secure. By its very nature, an encryption system is difficult to test. In practice, "security" only means that the system has been subjected to careful analysis and repeated attack and has remained unbroken.

We describe our system in detail sufficient to allow its implementation on most microcomputers. We also make our design choices explicit and offer argument to support them, so that readers can form their own opinions as to the strength of the system. We make no claim of theoretical breakthrough here, only the careful application of cryptographic principles. Finally, we offer the design and program code to the public, free of any claim of copyright or proprietary right. If our system can withstand critical analysis, then the public will have an encryption system it can trust.

Some Cryptographic Basics

A few definitions are appropriate first. We use the term "encryption" to refer to the general process of making plain information secret and making secret information plain. To "encipher" a file is to transform the information in the file so that it is no longer directly intelligible. The file is then said to be in "ciphertext." To "decipher" a file is to transform it so that it is directly intelligible; that is, to recover the "plaintext."

The two general devices of encryption are "ciphers" and "codes." A cipher works on the individual letters of an alphabet, while a code operates on some higher semantic level, such as whole words or phrases. Cipher systems may work by transposition (shuffling the characters in a message into some new order), by substitution (exchanging each character in the message for a different character according to some rule), or by a combination of both. In modern usage, transposition is often called "permutation."

Shannon referred to substitution as "confusion" because the output is a nonlinear function of the input, thus creating confusion as to the set of input characters. He referred to transposition as "diffusion" because it spreads the dependence of the output from a small number of input positions to a larger number.[1]

A cipher that employs both transposition and substitution is called a "product" cipher. In general, product ciphers are stronger than those using transposition or substitution alone.

Every encryption system has two essential parts: an algorithm for enciphering and deciphering and a "key" that consists of information to be combined with the plaintext according to the dictates of the algorithm. In any modern encryption system, the algorithm is assumed to be known to an opponent, and the security of the system rests entirely in the secrecy of the key.

Our goal is to translate the language of the plaintext to a new language that cannot convey meaning without the additional information in the key. Those familiar with the concept of "entropy" in physics may be surprised to learn that it is also useful in information theory and cryptography. Entropy is a measure of the amount of disorder in a physical system or the relative absence of information in a communication system.

A natural language such as English has a low entropy because of its redundancies and statistical regularities. Even if many of the characters in a sentence are missing or garbled, we can usually make a good guess as to its meaning. In contrast, we want the language of our ciphertext to have as high an entropy as possible; ideally, it should be utterly random.

Our guiding principle is that we must increase the uncertainty of the cryptanalysts as much as possible. Their uncertainty should be so great that they cannot make any meaningful statements about the plaintext after examining the ciphertext; they must be just as uncertain about the key, even if they have the plaintext itself with the corresponding ciphertext (in practice, it is impossible to keep all plaintext out of their hands).

A prime consideration in the security of an encryption system is the length of the key. If a short key (short compared with the length of the plaintext) is used, the statistical properties of the language will begin to "show through" in

by John A. Thomas and Joan Thersites

John A. Thomas, 621 N.W. Sixth Street, Grand Prairie, TX 75050.

Copyright © 1984 John A. Thomas and Joan Thersites. All rights reserved.

the ciphertext as the key is used over and over; cryptanalysts will be able to derive the key if they have enough ciphertext to work with. On the other hand, we want a relatively short key so that it can be easily stored or even remembered by a human.

One way to meet these conflicting requirements is to use a short key to generate (or cause to be generated) an internal key of greater length, which is then used for the actual encryption. The trade-off here is that the long key must be generated by software, and it would seem that any key-generating software could be worked backwards to produce the rest of a key sequence if part of it is known. We will see that this is not necessarily true.

The other important fact about the keys is that we need a huge number of them. If our system allows only 10,000 different keys, for example, it is not secure. Our opponents could try every possible key in a reasonable amount of time. Fortunately, methods of generating a long internal key from a short external key also lend themselves to the production of a large number of different keys.

Thus, we want our system to be strong, fast, reasonably easy to code, and flexible enough to permit us to trade off speed for security or the opposite. The system we propose should satisfy the needs of the individual user who is concerned primarily with the storage of information and not with its high-speed transmission in large volumes. We assume also that the encryption is being carried out on the individual's own computer and not by a program running on some remote system.

Some Competing Systems

With these general considerations in mind, let's choose a specific encryption scheme for small systems. Three candidates come to mind: the government-sponsored Data Encryption Standard (DES), the new public key systems, or some approach to the infinite key cipher.

The DES is a product cipher using 16 stages of permutation and substitution on blocks of 64 bits each. The permutation tables are fixed, and the substitutions are determined by bits from a 56-bit key.[2] This short key has caused some experts to question the security of DES.[3] Also, because the system was originally intended for hardware implementation, it would be slow in software.

We are inclined to reject DES merely because it is a standard. A standard cannot be changed for convenience—to use, say, only part of the transformations to speed up processing or a longer key for additional security. The individual user is better off without a standard, since breaking an unknown custom-built cipher presents greater difficulties for the cryptanalyst.

The public key cipher is an interesting new development that shows potential for making other encryption systems obsolete. It takes its name from the fact that the key information is divided into two parts, one of which can be made public. A person with the public key can encipher messages, but only someone with the private key can decipher them.[4]

All of the public key systems rely on certain functions for which the inverse is very difficult to compute without the information in the private key. These schemes do not appear to be practical for microcomputers, at least for 8-bit machines, if their strength is to be fully exploited.[5] One variety of public key system, in fact, has been broken by solving its enciphering function,[6] but this is no reflection on other systems, such as the RSA scheme,[7] which use different enciphering functions.

We believe that the most practical encryption method for small systems is a product cipher in which the substitution mechanism as nearly as possible approximates the infinite key cipher.

The infinite key cipher, also known as the Vernam system or the "one-time pad," is simple in concept. We first generate a key that is random and at least the same length as our message. Then, for each byte of plaintext, we add the corresponding byte of the key to give the ciphertext. By "add" we mean some reversible operation; the usual choice is the exclusive-or.

A little reflection will show that, given a random key at least the size of the plaintext (i.e., "infinite" with respect to the plaintext because it is never repeated), the resulting cipher is unbreakable, even in principle.[8]

This scheme is in use today for the most secret government communications, but it presents a serious practical problem by requiring not only a long random key for each message but also that the lengthy key be sent to the recipient. Thus, the ideal infinite key system is not practical for large volumes of message traffic.

A Practical Infinite Key Cipher

We can design a substitution method that will approximate an infinite key by generating the key as we go. Of course, any deterministic method of generating a key has, in principle, some regularity that an analyst might exploit. However, if we make the "work factor" for breaking our system so high that it is impractical for our opponents to do so, then it is irrelevant that the system may be less strong than the ideal.

What constitutes an adequate work factor depends essentially on the number of uncertainties the cryptanalysts must resolve before they can derive plaintext or a key. In these days of constantly improving computers, that number should probably exceed 2^{128}.

It is easy to quantify the work factor if we are talking about exhaustive key trial, but few modern ciphers are likely to be broken by key trial since it is too easy to make the key prohibitively large. Most likely they will be broken because of internal periodicities and a subtle dependency of output on input; these give the cryptanalysts enough information to reduce their uncertainty by orders of magnitude.

Since increasing the work factor for the cryptanalysts also increases it for the encryption program, we want to build a modular system. That way we can add or remove transformation functions to trade off time for security, thus meeting whatever threat we reasonably expect.

We start with the problem of key length. How can we obtain a key effectively as long as the plaintext without having to actually specify it, byte by byte? The answer is key expansion.

Suppose we have three short keys, SECRECY (length 7), COMPUTER (length 9), and MATHEMATICS (length 11). We obtain a longer key by the exclusive-or of these three together, byte by byte. That is, S XOR C XOR M equals the

first key byte; E XOR O XOR A equals the second key byte, and so on. If we reach the end of a short key, we start over at its first character. If the key lengths are relatively prime (that is, they have no common divisor greater than one), then the period of the composite key is equal to the product of their periods (7 x 9 x 11 = 693).

Obviously, we can generate a composite key of any length we want by increasing the length of the keys and by increasing their number. A useful article by Sclawy discusses this method and describes a version of it for ASCII files in BASIC.[9]

Key expansion is not a free ride, however; the effective length of the composite key, from the cryptanalytic point of view, is actually not as great as the product of the short keys because periodicities may appear in the key stream due to repeated traversal of the short keys. We can alleviate this weakness by simply increasing the size and number of our short keys. We can also obscure these periodicities by adding transposition.

Readers familiar with the history of cryptography may notice a resemblance between our short keys and the rotors of the mechanical cipher machines of World War II, such as the German ENIGMA. The rotor devices are relatively strong, but they can be broken, and we have taken their weaknesses into account in designing our system. Henceforth, we will usually refer to the short internal keys of our system as "rotors."

We now construct a toy encryption system based on this principle. If we also want to add permutation to our cipher, we must choose some unit or "block" of plaintext over which the permutation will take place. Let's say we choose a six-character block. Then we can also perform our substitutions over the same number of characters, advancing the rotor pointers by six after each block is processed.

Figure 1 (below) shows the tables involved. Note the three rotors carefully. They are of relatively prime lengths (19, 23, and 31), but the first six characters of each rotor of length n have been spliced onto the end of the rotor at position $n+1$.

Thus, we simply add six to the rotor pointer after each block, instead of after every character. This arithmetic is modulo the rotor length, so we reset the pointers to the beginning of the rotor after the sum of the pointer plus six exceeds the rotor length. We call the rotor pointers the "starts," since they point to the starting position for the substitution.

Note also the table of numbers in 0..5 called "PERM." This is the permutation table, and it determines how the characters in the block are to be transposed. In this example, the number 5 is in the 0th position, so character 5 in the block goes to the 0th place; 0 is in the 1st position, so character 0 goes to the 1st position, and so on until arriving at the permuted result. Here, if the input block is ABCDEF, then after permutation the block will be FABDEC.

Leaving aside for the moment how PERM *and the rotors* are constructed, we can sketch a pseudo-code outline of our enciphering function in Listing One (page 54). To decipher, we submit the ciphertext to substitution and then perform the inverse permutation. Note that deciphering is not the inverse of enciphering; we must know which way we are going.

With a larger block size and a few more rotors, this is essentially the ENIGMA machine. It is actually a little stronger, since ENIGMA applied the inverse permutation after the substitutions, which made deciphering the inverse of enciphering.

ROTOR0 LEN = 19	ROTOR1 LEN = 23	ROTOR2 LEN = 31	PERM
A	E	C	5
V	W	A	0
M	X	E	1
C	P	I	3
X	A	Z	4
L	L	Q	2
-	-	-	
Q	W <-START1	E	
T	Y	U	
I	O	P	
A	S	D	
F	G	H	
J <- START0	K	L	
-	-	-	
M	L	K	
J	H	G	
F	D	S	
A	M	N	
B	V	C	
X	Z	P	
-	-	-	
E	W	Q	
= splice	R	H	
A	X	M	
V	G	H	
M	D	J	
C	= splice	A	
X	E	-	
L	W	I	
	X	H	
	P	G	
	A	E	
	L	W	
		M	
		-	
		A	
		= splice	
		C	
		A	
		E	
		I	
		Z	
		Q <- START2	

START POSITIONS SHOWN AFTER ENCRYPTION OF 5 BLOCKS

Figure 1

Making the Practical Cipher Strong

Let's turn this toy system into a strong one. The most obvious enhancements are an increase in the number of rotors and the addition of more stages of permutation, consistent with a reasonable processing time. We chose six rotors and two stages of permutation for our system. The permutation occurs after the second and fourth substitutions.

Next, we determine that our system will operate on bits instead of bytes, for both substitution and permutation. (This is called "fractionation," and ciphers using it have historically been strong ones.) We take 256 bits (32 bytes) as our block size and perform all substitutions and permutations on such blocks.

We also want our rotors to be relatively large to achieve a long period, so we choose rotors whose lengths, in bits, are prime numbers less than 2048 (256 bytes). Specifically, we choose rotors of lengths 1789, 1787, 1783, 1777, 1759, and 1753, giving a composite period of 3.1×10^{19} bits. Choosing

prime numbers for the rotor lengths is an easy way to assure relative primality.

There is no reason to apply the same permutation to each block if PERM can take on a different arrangement for each block. As it turns out, a permutation table can cycle through a number of different arrangements before repeating itself. Our permutation table, now called BITPERM, consists of 256 randomly ordered bytes in 0..255, each byte value corresponding to a particular bit in the block. After each block, we will "cycle" BITPERM (see below) to a new position, allowing us to process 256 32-byte blocks before the same permutation is used again. This increases the period of the key stream to 7.9×10^{21} bits.

In our example, the rotors always step six characters at a time. We can easily provide for the rotors to step some arbitrary distance (modulo the rotor size). Our opponents now must contend with the fact that each rotor may be stepping anywhere from 1 bit to (rotor size–1) bits at a time. Also, we can change the order of the rotors for each session, giving 90 possible significant orderings.

We have drastically increased the uncertainties confronting our opponents, but we must add one further refinement. Even though our key stream is very long, we will use only the first tiny fraction of it—even for huge files. This is not only wasteful but dangerous. If the key stream is always set to the same starting point in its sequence, then identical plaintext files will produce identical ciphertext files. Even statistically similar files will produce similar ciphertext files. This gives the cryptanalysts more information about the key than we want them to know.

In principle, collecting enough such ciphertext will enable them eventually to reconstruct the rotors. To explode this possibility, we count the number of blocks we have enciphered and keep this information. Before each new session, we check this index and begin enciphering or deciphering from that point on. We write out the index as the first record of the ciphertext file and read it before deciphering; if enciphering, we save it in a separate file.

In our system, the index record is one CP/M record, with the first nine bytes reserved for the 72-bit index and the remaining bytes unused. The index thus describes the starting point for each rotor and one cycle of the permutation table. For example, the particular setting of the permutation table will be the residue of the index value divided by 256 (e.g., the index modulo 256). The starting point for the rotor of length 1789 will be the index value times the chosen step value modulo 1789, and so forth.

If two persons are corresponding, one could take all the blocks starting at block number 100 million, and the other could take all blocks starting at 1000 million. Even though they might send many identical message strings, the key stream used would always be different. In this way, we approach the requirement of the ideal infinite key cipher: that the key be used only once.

Note that we do not encipher the index record. We would have to do so at some agreed-upon index value to recover it, and this block would be nearly the same from message to message, again giving the cryptanalysts too much information. We follow the cryptographic principle: that which cannot be deeply hidden should be sent in the clear. This tells the cryptanalysts nothing they would not know if we did not use an index, since they would then know from the algorithm itself what the starting points were.

Constructing the Keys

This design implies that we must construct six random tables of bits for the rotors, with the initial 256 bits spliced onto the end of each rotor. (We designed our rotors so that the rotor, plus the splice, occupies 256 bytes or less.) We must also construct the permutation table of 256 random bytes.

In a sense, these tables constitute our key, since the key stream used for encryption is derived from them. It is obviously impractical for us to enter the tables each time we wish to send or read a message. We would prefer to enter some manageable "user key" that would cause the tables to be generated. We could, of course, have no user key at all and just make the tables a fixed part of the algorithm. But eventually the tables would be compromised, and they are not easily changed.

A convenient solution, which we carefully considered, would be to use a software random-number generator to create the tables. The user key could then be the seed for this generator. Common ways of generating random numbers, such as the linear congruential method, are constructed so that the next random number is generated by transforming the preceding random number.[10] If one part of the sequence is known or correctly guessed, then all of the sequence can be computed.

We believe that this can be circumvented by constructing a generator of very large word size, say 96 bits, and using only the most significant 8 bits for the output. We do not see how such a generator could be worked backwards, but a comment by Knuth in one of his exercises suggests that it can be, at least for smaller word sizes.[11]

We finally decided to abandon linear random-number generators entirely when we realized that we had a good nonlinear generator in the encryption system itself. If we have already constructed a set of genuinely random tables, we can take the user key, or a part of it, as a plaintext block that we then encipher using the predetermined rotors. The enciphered block replaces the first 32 bytes of the first rotor, and we encipher the just-enciphered block again; this time, however, the first rotor is different, having just been altered. We place the newly enciphered block in the second 32 bytes of the rotor and continue until we have filled all six rotors and the permutation table, thus replacing them entirely with new random values.

This self-modifying feature makes the new tables, which we call the "pro tempore" tables, depend on the user key in a very complex way. The pro tempore tables are the ones actually used for enciphering and deciphering, after having been properly spliced.

To guarantee that the output of this process is random, we must be sure that the predetermined tables are also random. Here is a cheap but effective method of doing so. To create the permutation table, begin with a bag of lima beans, a fine-point indelible ink pen, a bowl, a fork, and a tablet of paper. Carefully number 256 beans with the digits 0 through 255. As you number the beans, lay them out systematically before you on a flat surface in 16 rows of 16 columns each. This way

you can check that all numbers from 0 to 255 are present and that no numbers are duplicated. Place the numbered beans in the bowl and stir well with the fork. Without looking in the bowl, select beans one by one and record their numbers on the pad. Do not put the beans back in the bowl. The resulting table is the *predetermined* permutation table, which we call RANDP in our program.

The procedure for constructing the rotors is similar, but this time you must mark 64 beans with 1 on one side and 0 on the other, like the heads and tails of a coin. Stir these 64 beans with the fork and, again without looking, draw them out one by one and place them in a row. Record the resulting bit pattern and put the beans back in the bowl. Stir well again and repeat the process until you have all of the bits for the rotor in question.

Take the first 256 bits and write them down following the rotor bits. Divide the bit stream up into blocks of 8 bits and compute the hexadecimal value of each. This will give you a table of hex values that you can enter with an editor. Repeat this process until you have constructed all six rotors. This is tedious and requires meticulous work, but you must do it to guarantee genuinely random tables.

We decided upon three user keys, each of which may be between 1 and 75 ASCII characters in length. Since there are 95 printable ASCII characters, the user keys with the ASCII bias removed are considered as numbers in base 95 notation. We convert these numbers to binary (base 2) and pass them to the encryption functions.

We use one of the user keys as the initial plaintext in the table-generation process described above. The other user keys determine the stepping distances and the starting positions for the rotors during the generation phase and the stepping distances for the encryption session. The starting points for the session, of course, are determined by the index record.

The Program

We wrote our program in Z80 assembly language, using the Digital Research RMAC relocating assembler. This assembler uses the Intel mnemonics for the 8080 instructions and macros for the Z80 instructions. Where the macros differ from the Zilog mnemonics, we supply a cross reference.

We would have preferred to write these routines in a high-level language, but the amount of time taken by the intensive bit manipulation precluded that. The listings that follow describe the encryption functions only; we omit the part of the program concerned with collecting the user keys and file I/O, since this code is straightforward.

First consider the module PARAMS (Listing Two, page 55). Here are the six rotors, the permutation tables, and storage for the starts, steps, rotor lengths (hereafter called the "rotor moduli" because of their use in computing the starts and steps), and user keys. The table I6, which determines the order in which the rotors are taken, will be shuffled into some random permutation. We extract 32 bytes of text from the file buffer, place it in BLOCK to be enciphered or deciphered, and then move it back to the buffer.

Note that most of these data areas must be assembled and linked so as to reside on a page boundary. For this extra effort, we earn an appreciable gain in speed in the permutation routines.

Note also the three tables BITPERM, RANDP, and I256 BITPERM is the table consulted by the bit permutation routines, but it is uninitialized at startup. BITPERM is constructed by a routine called CYCLE (described below) before generation of the pro tempore rotors begins, and it is reconstructed from RANDP after each block. After the initialization process, RANDP is replaced, just as the predetermined rotors are, but it is replaced by shuffling *the ordered table* I256 according to 256 random bytes generated at initialization. We cannot simply fill RANDP with random bytes, since it can contain no duplicates.

KEYGEN (Listing Three, page 58) controls the startup process just described. It first calls UPTACK (described below) to convert the user key to a binary number and then reconstructs the tables by repeated encipherment. After KEYGEN finishes its work, we call INIT (Listing Four, page 66), which uses the index information to set the permutation table and the rotor starting positions to their correct places for this particular point in the key stream. We are finally

Statistical Test of Key Generator

Average of 20 trials of 5120 bytes each

	Normalized results	Theoretical values
Mean:	.49785	.50000
First moment:	.08351	.08333
Second moment:	-.00003	.00000
Third moment:	.01255	.01250

Chi-square Statistic

Probablity:	99%	95%	75%	50%	25%	5%	1%
Distribution:	187.2	205.5	233.4	254.3	276.5	310.3	335.9

Average of results: 258.9

Table

ready to encipher or decipher a file.

ENCIPHER (Listing Five, page 69) and DECIPHER (Listing Six, page 71) control the sequence of substitution and permutation on each block of text. Note that ENCIPHER takes the rotors in the order determined by I6. It calls EXTRACT to take 256 bits from the rotor in use, beginning at the current starting point, and to place them in the WORK common area. It calls SUBSF to perform the substitution on each block and calls PERMF to permute the bits in the block after the second and fourth substitution. Finally, it calls NXTBLK, which increments the index by one, steps the rotors to the next starting position, and sets BITPERM to its next permutation.

DECIPHER is similar. We take the rotors in the reverse order and call the inverse substitution and permutation functions, SUBSG and PERMG. (By mathematical convention, if "F" is a function, then "G" is its inverse.)

The substitution functions, SUBSF (Listing Seven, page 74) and SUBSG (Listing Eight, page 75) are simple. We chose to perform substitution modulo 2^{256} instead of modulo 2 (the exclusive-or); that is, we just add 32 bytes, discarding the *final carry*, for forward substitution and subtract 32 bytes without a final borrow for the reverse substitution.

With exclusive-or, cryptanalysts can narrow the possibilities for the key if they know the plaintext and ciphertext (i.e., if the ciphertext bit is 0 and the plaintext bit is 1, then the key bit must have been a 1, and so on). Addition mod 2^{256} spreads the uncertainty over the entire block, since the cryptanalysts don't know if there was a carry from byte to byte or not.

PERMF (Listing Nine, page 76) and PERMG (Listing Ten, page 78) are the bit permutation routines, PERMF being the forward permutation and PERMG being its inverse. These routines may interest the reader quite apart from their cryptographic use in our program. Since hardware designers have not seen fit to give us machines with bit addressing, manipulations at the bit level are awkward at best. Even so, bit permutation is not difficult, if you define the problem correctly.

These routines address bits in one special case: a bit stream where bit 0, the first leftmost bit, is aligned on a byte boundary. We can imagine the bits in the stream as being indexed with subscripts from 0 to n from left to right. The general subscript is i, so that we can talk of the ith bit, and i is always a base 2 integer.

The trick used in these routines is to imagine the bit stream not as a vector but as a matrix consisting of x rows and y columns. Every byte is one row beginning with row 0. Every y is a bit within a row. There are 8 bits within each row subscripted from 0 to 7, left to right. To address the ith bit, we convert the vector index i to the matrix coordinates x,y.

APL fans will recognize this as the "encode" function with index origin zero. The actual computation is almost effortless because the modulus of the column coordinate y is an integral power of 2, so we merely shift the binary integer i to obtain x and y, giving us the byte and bit coordinates of the desired bit.

For illustration, let i be 115 decimal, which is 01110011 binary. Imagine this number now as an ordered pair (x,y) with a demarcation after the third bit from the right:

```
  x    y
01110:011
```

To the left of the demarcation is the binary value of x, and to the right is the value of y; in other words, the 15th byte, 3rd bit, corresponds to the 115th bit. Since most computers allow byte addressing, and even bit addressing by one means or another within a byte, this trick permits general bit addressing with a minimum of computation.

We now describe what we mean by "forward" and "inverse" permutations. Suppose BITPERM contains the byte values 57, 15, 21, ... 173. The forward permutation

```
57  15  21  ...  175
 0   1   2  ...  255
```

means put the 57th bit in the 0th position, the 15th in the 1st, and so on to the 173rd bit in the 255th position. The inverse permutation means put the 0th bit in the 57th position, the 1st bit in the 15th, and so on.

The inverse permutation is about one-third faster than the forward permutation. This seems surprising until you realize that testing the bits for on or off is cheap in the inverse permutation but unavoidably expensive in the forward.

We have optimized PERMF and PERMG for speed, so they are not exactly models of structured programming. This is justified since most of the computation time required by TNT is spent in these two routines. Thus, we have required the operands for the permutation to be aligned on page boundaries to save instructions needed for byte addressing. This way, 8-bit registers alone can compute effective addresses. Also, we reach subroutines by jumps instead of by calls and returns.

CYCLE (Listing Eleven, page 81) is deceptively simple, but the concept behind it is abstract and perhaps unfamiliar to the reader; cryptographically it is critical. RANDP is a random permutation of the byte values 0..255, but RANDP is not a permutation list familiar to most programmers. It begins (in our example) 57, 15, 21, ...173. This does not mean "take the 57th element and put it in the 1st position." Rather, RANDP is a permutation in cyclic notation, denoted (57 15 21, ... 173).[12] This cyclic notation means "take the 15th element and put it in the 57th position, then take the 21st element and put it in the 15th, and so on until you take the 57th element and put it in the 173rd position."

For simplicity, and at the expense of some cryptographic strength, we have chosen one cycle of degree 256—that is what CYCLE is designed for. But CYCLE could be extended to handle N cycles of different degrees. Some good two-cycle lengths for TNT are 148 and 107, 167 and 87, and so on. These lengths are primes whose sum is 256. A good four-cycle sequence is 83 73 53 47. Again, these are primes that add up to 256.

Were we to use two or more cycles, say m and n, against RANDP, we would take the first m elements of RANDP as the first cycle, the next n elements as the second, and so on. For example, if our cyclic sequence were 83 73 53 47, the first 83 elements would be the first cycle, the next 73 elements the second, the next 53 the third, and the last 47 the fourth.

Why use cyclic notation? The cryptographic advantages in control and key length are great. The "order" of a permutation is defined as the least common multiple of the degrees of its cycles. The order means how many times the same permutation can be used to repeatedly permute something before that something permutes back to its starting arrangement. That is the whole idea cryptographically: we want to use the same thing (a permutation) repeatedly, but each time we want a different result. We also need to quickly compute what the Nth arrangement will be so that we can continue encryption of the next file where we left off from the previous file. Cyclic notation is perfect for this.

CYCLE uses RANDP to generate 256 different BITPERMs. If we used the cycles 149 and 107, we would generate 15,943 BITPERMs. By using more permutations in TNT, with well-chosen cycles, and staggering them, we could achieve thorough bit mixing with long periods.

We have restricted CYCLE to handle only one cycle of 256 elements, since we can quickly compute the position of the cycle by taking the index mod 256, which is just the least significant byte of the index. More time would be needed if we had to compute the residue of several prime cycles. Once we get the cycle position, we use it to shuffle RANDP into BITPERM. If the result is 0, then byte 57 (the 0th byte) of RANDP goes to byte 57 of BITPERM, byte 15 of RANDP goes to byte 15 of BITPERM, and so on. If the result is 1, then byte 15 (the 1st byte) of RANDP goes to byte 57 of BITPERM, byte 21 of RANDP goes to byte 15 of BITPERM, and so on.

UPTACK (Listing Twelve, page 82) converts a number represented in one base to a base 2 number. ("Uptack" is the informal name for the APL "decode" function.) Its argument is a byte string whose elements represent the number *n* in any base from 2 to 255. For example, suppose *n* is a base 10 number consisting of *m* digits. Then the argument is *m* bytes consisting of binary values ranging from 0 to 9. UPTACK's second argument is the base. In TNT the base is always 95, but we have written the routine to handle the general case.

The algorithm is simple number theory. First zero an accumulator. Then add the most significant digit to it. If there is another digit of *n*, multiply the accumulator by the base and add the next digit in. Continue until the digits of *n* are exhausted.

UPTACK has a subtle purpose: it is meant to make the use and maintenance of the user keys easy. Key maintenance and security are dependent on the user's resources. The most secure keys are those that are utterly random, but such keys cannot possibly be remembered and so must be recorded. This means that the user must provide physical security to guard the keys, an expensive business possible for governments and large corporations but not for the average citizen. Individual users cannot have recorded keys and still feel secure since such keys can be stolen or seized under legal process.

We therefore want keys that can be easily memorized. Any key that can be memorized, however, tends to be cryptographically weak; your opponent may guess it or find it by exhaustive search. Suppose you use an English word of eight characters or fewer. It would be easy to try all such words from a dictionary in a reasonable amount of time.

A good user key, then, should be long, have no connection with you, and not be available to the opponent in any published work. Poetry is fine, but compose it yourself! UPTACK

```
                    Statistical Test of Ciphertext

                Average of 15 trials of 5120 bytes each

                      Normalized results      Theoretical values

Mean:                      .50102                  .50000
Second moment:             .08327                  .08333
Third moment:              .00016                  .00000
Fourth moment:             .01247                  .01250

                          Chi-Square Statistic

Probability:    99%     95%     75%     50%     25%     5%      1%
Distribution:  187.2   205.5   233.4   254.3   276.5   310.3   335.9

Average of results:  254.2
Range:  206.8 to 292.2

                              Running Times

   Initialization (including read of index):
           4.8 seconds

   Enciphering (exclusive of disk i/o) for 31 CP/M records:
           6.5 seconds, or 610 bytes/second

   Deciphering (exclusive of disk i/o) for 31 CP/M records:
           5.0 seconds, or 793 bytes/second
```

Figure 2

helps out here because it allows you to think of a key in human terms, even though it is just a random bit stream to the encryption program. Protection of the user keys is so important that most of the TNT code is occupied with it.

The other support routines, not shown, are:

(1) EXTRACT, which extracts 256 bits from a rotor beginning with the *n*th bit to the *n*th + 256. On call, HL addresses the bit stream input on a page boundary, and DE holds *n*. On return, HL addresses the extracted bits aligned on a byte boundary in the WORK common area.

(2) RESIDUE, which returns the residue of an 88-bit number divided by a 16-bit number. On call, DE holds the divisor, and HL addresses the dividend. On return, DE holds the result.

(3) SHUFFLE, which sets up the permutation table, BITPERM, by shuffling the ordered array, I256. On call, BC addresses RANDP, DE addresses the workspace BITPERM, and HL addresses I256. On return, DE addresses BITPERM.

(4) SPLICE, which splices the first 256 bits of a rotor onto the rotor after the $n-1$ bit of the rotor. On call, BC holds n (the rotor size), and HL addresses the rotor.

(5) NXTBLK, which prepares the program for encryption of the next block. It advances the index value by one, advances each start by its corresponding step, and sets BITPERM to its next cycle.

(6) RIPL0, which merely sets the WORK common area to zero.

Conclusion

At the beginning of this project we wondered whether a program requiring so much bit manipulation would be practical on an 8-bit microprocessor. We are pleased to report that the program runs rapidly and could probably be strengthened further without an unreasonable increase in processing time.

Figure 2 (page 52) summarizes the results with TNT running on a 4-MHz Z80. About half of the total time is used by the bit permutation routines. We also tested the ciphertext files for statistical randomness, using the methods suggested by Ruckdeschel.[13] Figure 2 shows the results of several trials. The output appears to be completely random and independent of the input, even with plaintext files containing only zeroes or only ones. You must understand that a random output is only a *necessary* requirement for a strong encryption system, not a sufficient one, but we are delighted with the results anyway.

We feel we have satisfied the basic requirements for a strong encryption system:

(1) A large key space
(2) A long, practically infinite key period
(3) A random key stream and a random ciphertext
(4) No apparent dependency of output on input
(5) A good combination of substitution and permutation
(6) A user key that is convenient but not easily compromised

We can think of several ways TNT could be improved. Huffman encoding of the plaintext files before enciphering would provide a substantial increase in speed for large files. For greater cryptographic strength, consider the improvements to the permutation routines suggested in the discussion of CYCLE. More rotors are always good, if you can stand the extra time. With a 16-bit machine, the possibilities get even more interesting. If you decide to implement TNT yourself, construct your own tables.

For readers interested in seeing TNT run, we will send a standard 8-inch single-density disk containing the object code, all source files, and documentation to those who send $25. It is traditional in introducing a new encryption system to issue a challenge cipher, and we include one on the disk. Our plaintext is an excerpt from a news story discussing government threats that persons interested in cryptography had better cooperate with the intelligence agencies or else.[14] We present the corresponding ciphertext, which, of course, includes the index record, and we challenge the reader to solve for the pro tempore rotors and RANDP. For a bonus, determine the natural language user keys. This provides you with everything a cryptanalyst could hope for, but if TNT cannot resist solution here, it will not have much practical value. We welcome your comments and suggestions.

References

1. Shannon, C., "Communication Theory of Secrecy Systems," *Bell System Technical Journal* (October 1949): 656-715.
2. Katzan, H., *The Standard Data Encryption Algorithm*, Petrocelli Books, 1977.
3. Diffie, W., and Hellman, M., "Exhaustive Cryptanalysis of the NBS Data Encryption Standard," *Computer 10* (June 1977): 74-84.
4. Hellman, M., "The Mathematics of Public Key Cryptography," *Scientific American*, Vol. 241, No. 16 (August 1979): 146-157.
5. Smith, J., "Public Key Cryptography," *Byte* (January 1983): 198-218.
6. "Adelman and Apple II Bust Public-Key Crypto Code," *Infoworld*, October 4, 1982.
7. Rivest, R., et al., "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Communications of the ACM*, Vol. 21, No. 2 (February 1978): 120-126.
8. Feistel, H., "Cryptography and Computer Privacy," *Scientific American*, Vol. 228, No. 5 (May 1973): 15-24.
9. Sclawy, A., "CP/M Encryption Prescription," *Microcomputer* (November 1980): 42-46.
10. Knuth, D., *The Art of Computer Programming*, Vol. 2, 2nd ed., Addison-Wesley, 1981.
11. Knuth, Vol 2, page 177.
12. Knuth, D., *The Art of Computer Programming*, Vol. 1, 2nd ed., Addison-Wesley, 1973.
13. Ruckdeschel, F., "Testing a Random Number Generator," in *Numbers in Theory and Practice*, p. 121, Byte Publications, 1979.
14. "CIA Director Warns Scientists," *Science*, January 22, 1982.

■■J

(Listings begin on page 54)

File Encryption (Text begins on page 44)

Listing One

```
/*************************************************************************
*                                                                        *
*  PSEUDO-CODE OUTLINE OF SIMPLE ENCIPHERING FUNCTION.                   *
*  WE ASSUME:                                                            *
*       1. BLOCK, PERM, ROTORS AND STARTS ARE EXTERNAL                   *
*          TO THIS PROCEDURE,                                            *
*       2. BLOCK, PERM AND THE ROTORS ARE ALREADY                        *
*          CONSTRUCTED, AND,                                             *
*       3. THE STARTS ARE INITIALIZED.                                   *
*                                                                        *
*                                                                        *
*************************************************************************/

ENCIPHER:
    PROCEDURE(BLOCK);

    DECLARE
        BLOCK[6]     CHAR,              /* THE INPUT TEXT */

        PERM[6]      CHAR,              /* THE PERMUTATION TABLE */
        ROTOR0[25]   CHAR,              /* THE SPLICED ROTORS */
        ROTOR1[29]   CHAR,
        ROTOR2[35]   CHAR,

        LEN0         INT INIT(19),      /* THE ROTOR LENGTHS */
        LEN1         INT INIT(23),
        LEN2         INT INIT(29),

        START0       INT,               /* ROTOR STARTING POINTS */
        START1       INT,
        START2       INT,

        WORK[6]      CHAR,              /* SCRATCH SPACE */
        I            INT;               /* INDEX */

    /* PERFORM THE PERMUTATION */

    WORK = BLOCK;                       /* COPY BLOCK TO WORK */
    FOR I = 0 TO 5;
        WORK[I] = BLOCK[PERM[I]];       /* EXCHANGE EACH CHAR */
    END FOR;
    BLOCK = WORK;                       /* PLACE RESULT IN BLOCK */

    /*
       PERFORM SUBSTITUTION WITH EACH ROTOR.
       HERE, "+" REPRESENTS ADDITION MODULO SOME N FOR
       THE 6 CHARACTERS; IF N = 2 THEN THIS IS THE
       EXCLUSIVE-OR.
    */

        BLOCK = BLOCK + ROTOR0[START0];
        BLOCK = BLOCK + ROTOR1[START1];
        BLOCK = BLOCK + ROTOR2[START2];

    /* ADVANCE THE STARTING VALUES FOR NEXT BLOCK */
```

```
    START0 = (START0 + 6) MOD LEN0;
    START1 = (START1 + 6) MOD LEN1;
    START2 = (START2 + 6) MOD LEN2;

    RETURN (BLOCK);

END ENCIPHER;
```

End Listing One

Listing Two

```
;*****************************************************************;
;                                                                 ;
;  PARAMS:  A block of storage for the various parameters         ;
;       used by the encryption program.   This block must         ;
;       be linked so as to reside on a page boundary.             ;
;           Most of these areas are accessed by pointers, not     ;
;       by their common names; the "common" definition is only    ;
;       to allow page alignment.                                  ;
;                                                                 ;
;*****************************************************************;

        DSEG

        COMMON  /BLOCK/
BLOCK   DS      32              ;holds the block of text to be
                                ;...transformed by the encryption
                                ;...process.

        DS      224             ;skip to next page

        COMMON  /WORK/
WORK    DS      33              ;work space used by various
                                ;..routines
        DS      223             ;skip to next page

                                ;the following rotors consist of
                                ;..bits generated by a process
                                ;..known to be truly random
                                ;..with replacement
        COMMON  /ROTOR0/
ROTOR0  DB      107,97,132,125,141,82,167,197,158,229,166,29,66
        DB      179,22,139,240,90,117,254,137,142,130,224,162,53
        DB      155,104,114,51,142,244,23,43,146,187,189,134,42
        DB      49,163,91,152,141,151,67,111,136,135,230,136,120
        DB      31,236,131,159,69,94,173,107,66,105,22,109,2,235
        DB      124,232,165,228,212,45,213,141,195,13,140,149,105
        DB      158,165,206,91,192,210,75,110,216,175,198,250,0
        DB      181,46,169,208,194,114,238,88,210,250,145,50,106
        DB      247,128,139,61,132,230,83,7,246,66,133,244,197,46
        DB      103,243,212,170,39,72,157,18,118,198,88,66,187,36
        DB      88,239,217,232,5,131,51,199,107,88,175,166,20,58
        DB      255,81,204,70,83,150,109,210,120,165,248,38,49,180
        DB      112,237,148,245,107,32,23,82,130,223,124,226,228,129
        DB      129,61,210,56,98,52,243,176,142,219,123,9,7,25,11
        DB      127,23,166,109,203,139,59,208,136,23,102,116,112
        DB      121,237,29,9,138,122,20,92,188,31,153,79,102,52,136
        DB      178,165,12,221,56,254,203,91,12,35,236,106,149,62
        DB      44,247,45,48,234,21,152,180,95,130,211,175,244,76
        DB      116,23,5,17,172,219,67,145,156,119,160
```

(continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Two

```
                                    ;The remaining rotors are not
                                    ;..shown completely.  Of course,
                                    ;..each has 256 random values.
        COMMON  /ROTOR1/
ROTOR1  DB      177,238,109,128,55,20,169,122,87,56,108,47,166
        DB      113,212,11,81,246,185,69,33,29,103,77,135,110
        .
        .
        DB      13,133,244,206,58,129,106,62,215,40,164,35,172,233
        DB      176,237,199,70,18,172,21,61,255

        COMMON  /ROTOR2/
ROTOR2  DB      104,166,98,251,237,91,132,51,97,57,121,212,210,235
        DB      13,215,104,176,75,144,87,76,42,169,197,11,224,165
        .
        .
        DB      96,151,32,174,152,85,83,138,23,193,75,243,156,119
        DB      184,237

        COMMON  /ROTOR3/
ROTOR3  DB      178,17,130,147,234,183,242,38,221,88,255,36,234
        DB      141,236,112,126,233,175,188,188,89,22,161,151,89
        .
        .
        DB      249,19,110,172,127,146,117,70,246,56,63,116,215,222
        DB      94,44,139,80,203,172,227,41,157,231,39,237,61,169

        COMMON  /ROTOR4/
ROTOR4  DB      145,99,204,93,181,0,29,191,66,104,100,107,152
        DB      249,180,183,190,159,84,59,206,225,60,115,203
        .
        .
        DB      157,194,120,231,150,15,20,58,164,220,1,100,141
        DB      55,238,238

        COMMON  /ROTOR5/
ROTOR5  DB      45,76,195,139,23,99,34,35,233,59,81,46,240,236
        DB      138,228,25,112,181,239,189,37,232,156,188,21,178
        .
        .
        DB      222,146,244,78,94,10,217,33,219,203,133,133,211,38
        DB      58,139,20
                                    ;the following table describes
                                    ;..byte values generated by a
                                    ;..process known to be truly random,
                                    ;..without replacement
        COMMON  /RANDP/
RANDP   DB      57,15,21,125,40,138,197,120,3,74,232,251,239
        DB      82,49,121,208,142,170,195,198,206,237,233,66,90
        .
        .
        DB      199,154,177,56,98,217,126,97,116,210,70,222,203,43
        DB      11,100,157,28,79,178,6,145,155,99,181,169,253,173
        COMMON  /BITPERM/           ;holds the permutation table
BITPERM DS      256
```

```
                                    ;ordered array to shuffled
        COMMON  /I256/              ;..into BITPERM
I256    DB      0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18
        DB      19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34
        .
        .
        DB      243,244,245,246,247,248,249,250,251,252,253,254
        DB      255
                                    ;following areas need not be
                                    ;..aligned on page boundary
        COMMON  /I6/
I6      DB      0,1,2,3,4,5         ;ordered array to shuffle for
                                    ;..rotor choice
        COMMON  /GUARD/
GUARD   DB      0,0                 ;guard bytes for multiply overflow
        COMMON  /INDEX/             ;..from INDEX
INDEX   DS      9                   ;the file index, which counts the
                                    ;..number of blocks which have
                                    ;..been encrypted.  Initially it
                                    ;..holds a user key, which is used
                                    ;..to generate the tables for the
                                    ;..session.

        COMMON  /STEP0/
STEP0   DS      2                   ;the stepping distances for the
STEP1   DS      2                   ;..rotors
STEP2   DS      2
STEP3   DS      2
STEP4   DS      2
STEP5   DS      2

        COMMON  /START0/
START0  DS      2                   ;the starting bit for rotor
START1  DS      2                   ;..stepping and key extraction
START2  DS      2
START3  DS      2
START4  DS      2
START5  DS      2

        COMMON  /MOD0/              ;rotor moduli
MOD0    DW      1789
MOD1    DW      1787
MOD2    DW      1783
MOD3    DW      1777
MOD4    DW      1759
MOD5    DW      1753

        COMMON  /LENKY1/
LENKY1  DS      1                   ;stores length of user key entered
        COMMON  /KEY1/
KEY1    DS      75                  ;user key with ascii bias stripped
        COMMON  /LENKY2/
LENKY2  DS      1
        COMMON  /KEY2/
KEY2    DS      75
        COMMON  /LENKY3/
LENKY3  DS      1
        COMMON  /KEY3/
KEY3    DS      75
        END
```

End Listing Two

(Listing Three begins on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Three

```
;-----------------------------------------------------------------;
;                                                                 ;
;  KEYGEN:  Generate the pro tempore, or session keys, from       ;
;       the user keys.  The pro tempore keys are:                 ;
;           the six rotors, ROTOR0 - ROTOR5,                      ;
;           the permutation table, RANDP,                         ;
;           the rotor steping distances, STEP0 - STEP5,           ;
;           the rotor starting bits, START0 - START5, and,        ;
;           I6, the ordering index for rotors, starts,            ;
;               steps and moduli.                                 ;
;                                                                 ;
;       We use  LENKY1 and KEY1 to derive a tempory index         ;
;               and I6,                                           ;
;               LENKY2 and KEY2 to derive the rotors and the      ;
;               starts, and,                                      ;
;               LENKY3 and KEY3 to derive the steps and RANDP.    ;
;                                                                 ;
;       Macro           Zilog                                     ;
;       =====           =====                                     ;
;       LXIX nn         LD IX,nn                                  ;
;       JRC  nn         JR C,nn                                   ;
;       STX  r,n        LD (IX+n),r                               ;
;       INIX            INC IX                                    ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  KEYGEN
        EXTRN   UPTACK,RESIDUE,ENCIPHER,CYCLE,SHUFFLE,SPLICE
        EXTRN   IOBUFF

KEYGEN:
        PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H

        LDA     LENKY1          ;convert key1 to base 2
        MOV     B,A             ;length in B for UPTACK
        MVI     A,95            ;key is in base 95
        LXI     H,KEY1
        CALL    UPTACK
        LXI     D,KEY1          ;then move converted number
        LXI     B,32            ;..back to KEY1 space
        LDIR
        LDA     LENKY2          ;convert key2 in same manner
        MOV     B,A
        MVI     A,95
        LXI     H,KEY2
        CALL    UPTACK
        LXI     D,KEY2
        LXI     B,32
        LDIR
```

```
        LDA     LENKY3          ;..and key3 also
        MOV     B,A
        MVI     A,95
        LXI     H,KEY3
        CALL    UPTACK
        LXI     D,KEY3
        LXI     B,32
        LDIR

        LXI     D,GUARD+1       ;clear INDEX for use by RESIDUE
        LXI     H,GUARD
        XRA     A
        MOV     M,A
        LXI     B,10
        LDIR

                                ;set up the rotor starting bits
        LXIX    START0          ;..to whatever the user key
        LXI     H,KEY2+20       ;..specified, but adjusted to
        LXI     D,MOD0          ;..0 <= START[I] < MOD[I]
        CALL    GETRES
                                ;set up the rotor step values
        LXIX    STEP0           ;..to whatever the user key
        LXI     H,KEY3+20       ;..specified, similarily
        LXI     D,MOD0          ;..adjusted, but > 0
        CALL    GETRES
                                ;if any step = 0, force it to 1
        LXI     H,STEP0
        MVI     B,6

T01LOOP:
        MOV     E,M             ;get the two-byte step
        INX     H               ;..to DE
        MOV     D,M
        MOV     A,E             ;check it for zero
        ORA     D
        CZ      SETT01          ;set it to 1 if so
        INX     H
        DJNZ    T01LOOP
                                ;set up a temporary index
        LXI     D,INDEX
        LXI     H,KEY1+23
        LXI     B,9
        LDIR
                                ;set up BITPERM to Ith iteration
        LDA     INDEX+8         ;..by index mod 256
        LXI     H,RANDP
        LXI     D,BITPERM
        CALL    CYCLE

        ;-------------------------------------------------------;
        ;  The encryption parameters are now set up per the     ;
        ;  user keys.  We will use this setup as a random       ;
        ;  number generator to create the pro tempore rotors,   ;
        ;  permutation table, starts and steps actually used    ;
        ;  for the file encryption.                             ;
        ;-------------------------------------------------------;
```

(continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Three

```
        LXI     D,BLOCK         ;first, create the rotors
        LXI     H,KEY2          ;..by repeatedly enciphering
        LXI     B,32            ;..second user key
        LDIR
        LXI     D,ROTOR0
        MVI     B,48            ;do 48 blocks to fill the rotors

FILLROTORS:
        CALL    ENCIPHER        ;encipher the block
        LXI     H,BLOCK         ;..and move it to the rotor
        PUSH    B
        LXI     B,32
        LDIR                    ;leaves DE = DE + 32
        POP     B
        DJNZ    FILLROTORS
                                ;The rotors aren't completely
                                ;..fixed yet.   This must wait
                                ;..until I6 is generated,
                                ;..since it controls the order
                                ;..of rotors, etc.

        LXI     D,BLOCK         ;next generate pro tempore RANDP
        LXI     H,KEY3          ;move key3 to encryption block
        LXI     B,32
        LDIR
        LXI     D,IOBUFF        ;use i/o buffer as scratch space
        MVI     B,8             ;encipher 8 blocks

FILLRANDP:
        CALL    ENCIPHER        ;encipher the block
        LXI     H,BLOCK         ;..and move it to scratch area
        PUSH    B
        LXI     B,32
        LDIR
        POP     B
        DJNZ    FILLRANDP
                                ;call SHUFFLE to replace RANDP
        LXI     H,I256          ;address ordered table
        LXI     B,IOBUFF        ;address 256 random bytes
        LXI     D,RANDP
        CALL    SHUFFLE         ;now RANDP is different than
                                ;..the permanent RANDP


                                ;now generate I6
        LXI     D,BLOCK         ;move key1 to encryption block
        LXI     H,KEY1
        LXI     B,32
        LDIR
        LXI     D,IOBUFF        ;use i/o buffer as scratch space
        MVI     B,8             ;do 8 blocks
```

```
FILLI6:
        CALL    ENCIPHER        ;encipher block
        LXI     H,BLOCK         ;..and move it to scratch area
        PUSH    B
        LXI     B,32
        LDIR
        POP     B
        DJNZ    FILLI6
                                ;shuffle I256 to replace I6
        LXI     H,I256
        LXI     B,IOBUFF
        LXI     D,IOBUFF+256    ;shuffled I256 goes here
        CALL    SHUFFLE
                                ;now find the six bytes < 6
        LXI     D,I6
        LXI     H,IOBUFF+256
        MVI     B,6             ;loop control
SETI6:
        MOV     A,M             ;get a random byte
        CPI     6               ;is it < 6?
        JRC     IOK             ;jump if so
        INX     H               ;try another
        JMP     SETI6
IOK:
        STAX    D               ;save it in I6 if < 6
        INX     D               ;bump destination pointer
        INX     H               ;bump source pointer
        DJNZ    SETI6
                                ;finally, splice the rotors,
                                ;..in the order for this session,
                                ;..now that we have I6
        LXI     H,ROTOR0
        MVI     B,6             ;count 6 rotors, C indexes rotors

DOSPLICE:
        PUSH    B
        PUSH    H
        LXI     D,MOD0          ;address rotor moduli
        LXI     H,I6
        MOV     A,M             ;get I6 value
        ADD     A               ;* 2 for offset into moduli
        ADD     E               ;add to DE
        MOV     E,A
        JNC     DOSP1
        INR     D
DOSP1:
        XCHG                    ;HL -> the modulus to use
        MOV     C,M
        INX     H               ;get it to BC
        MOV     B,M
        POP     H               ;recover rotor address
        CALL    SPLICE          ;splice the rotor
        INR     H               ;point to next rotor (HL+256)
        POP     B
        DJNZ    DOSPLICE
                                ;now each ROTOR[I6[I]] is of
                                ;..length MOD[I6[I]], but in an
                                ;..order which depends on the
```

(Continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Three

```
                                    ;..user key
        POP     H                   ;all done...
        POP     D
        POP     B
        POP     PSW
        RET
                                    ;GETRES:  interface between KEYGEN
                                    ;..and RESIDUE.  IX -> destination,
                                    ;..HL -> source, DE -> mod, using
                                    ;..INDEX for dividend
GETRES:
        MVI     B,6                 ;loop control
RESLOOP:
        PUSH    B
        PUSH    D
        LXI     D,INDEX+7           ;move key bytes to INDEX
        LXI     B,2
        LDIR                        ;now HL -> next source
        POP     D
        XCHG                        ;HL -> mod
        PUSH    D
        MOV     E,M                 ;load modulus to DE
        INX     H
        MOV     D,M
        INX     H                   ;HL -> next mod
        PUSH    H
        LXI     H,GUARD
        CALL    RESIDUE             ;find the residue
        STX     E,0                 ;set destination to result
        STX     D,1
        INXIX                       ;next destination
        INXIX
        POP     D                   ;-> next mod
        POP     H                   ;-> next source
        POP     B
        DJNZ    RESLOOP
        RET

                                    ;SETTO1:  set step to 1 if req'd
SETTO1:
        MOV     M,A                 ;zero high byte (we know A is zero)
        INR     A                   ;make A = 1
        DCX     H                   ;-> low byte
        MOV     M,A                 ;store 1
        INX     H                   ;reset pointer
        RET

        COMMON  /BLOCK/             ;the encryption block
BLOCK   DS      32

        COMMON  /ROTORO/            ;start of the rotor space
ROTORO  DS      256
```

```
        COMMON  /RANDP/         ;permutation table
RANDP   DS      256
        COMMON  /BITPERM/       ;table used for bit permutation
BITPERM DS      256
        COMMON  /I256/          ;ordered table to be shuffled
I256    DS      256
        COMMON  /I6/            ;table for permutation of rotors
I6      DS      6

        COMMON  /GUARD/         ;guard bytes for index space
GUARD   DS      2
        COMMON  /INDEX/
INDEX   DS      9

        COMMON  /STEP0/         ;storage for the steps begins
STEP0   DS      2
        COMMON  /START0/        ;storage for the starts begins
START0  DS      2
        COMMON  /MOD0/          ;storage for rotor moduli begins
MOD0    DS      2

        COMMON  /LENKY1/        ;storage for user key lengths
LENKY1  DS      1
        COMMON  /KEY1/          ;..and user keys
KEY1    DS      75
        COMMON  /LENKY2/
LENKY2  DS      1
        COMMON  /KEY2/
KEY2    DS      75
        COMMON  /LENKY3/
LENKY3  DS      1
        COMMON  /KEY3/
KEY3    DS      75

        END
```

End Listing Three

(Listing Four begins on page 66)

(Continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Four

```
;-----------------------------------------------------------;
;                                                           ;
;  INIT:  Initialize the index from the file index          ;
;       (if enciphering), or from ciphertext file, if       ;
;       deciphering.  Also set BITPERM according to         ;
;       the index mod 256, and the rotor starting           ;
;       positions according to index and the steps.         ;
;       The index - 1 counts the the I - 1  block           ;
;       enciphered.                                         ;
;           This routine sets the pro tempore rotors        ;
;       and the bit permutation table to what they          ;
;       should be for enciphering the Ith block.            ;
;       Thus, no part of the key stream is used more        ;
;       than once.                                          ;
;                                                           ;
;       On call, HL -> encryption index                     ;
;                                                           ;
;       Macros          Zilog                               ;
;       ======          =====                               ;
;       RARR M          RR (HL)                             ;
;       JRC  nn         JR C,nn                             ;
;       RALR M          RL (HL)                             ;
;-----------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  INIT
        EXTRN   CYCLE,RESIDUE,RIPLO

INIT:
        PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H

        LXI     D,INDEX         ;destination area
        LXI     B,9
        LDIR                    ;INDEX = file or ciphertext index
        LXI     D,BITPERM
        LXI     H,RANDP
        LDA     INDEX+8
        CALL    CYCLE           ;BITPERM gets the permutation of
                                ;..RANDP per index mod 256

        ;-------------------------------------------;
        ;  Now set starting positions for rotors:   ;
        ;  DO I = 1 TO 6                            ;
        ;       START[I6[I]] = (INDEX * STEP[I6[I]])  ;
        ;                      mod MOD[I6[I]]       ;
        ;  END DO                                   ;
        ;-------------------------------------------;

        LXI     H,I6            ;-> indexes of START,STEP,MOD
        MVI     B,6             ;loop control
```

```
INITO1:
        MOV     A,M             ;get I6[I]
        ADD     A               ;offset from STARTO
        STA     OFFSET          ;save it
        INX     H               ;-> I6[I+1]
        PUSH    H               ;..and save for next loop
        LXI     D,STEPO
        ADD     E
        MOV     E,A
        JNC     INITO2
        INR     D
INITO2:
        CALL    MULT11          ;multiply index * STEP[I6[I]]
        LXI     H,MODO          ;get MOD[I6[I] into DE
        LDA     OFFSET          ;..for RESIDUE
        ADD     L
        MOV     L,A
        JNC     INITO4
        INR     H
INITO4:
        MOV     E,M
        INX     H
        MOV     D,M
        LXI     H,WORK+11       ;-> to 11 byte result of multiply
        CALL    RESIDUE         ;return with DE = residue

        LXI     H,STARTO        ;get START[I6[I]] to store
        LDA     OFFSET          ;..the residue
        ADD     L
        MOV     L,A
        JNC     INITO6
        INX     H
INITO6:
        MOV     M,E             ;HL -> START[I6[I]]
        INX     H               ;..store residue there
        MOV     M,D
        POP     H               ;retrieve I6[I+1]
        DJNZ    INITO1          ;do for I = 1 to 6

        POP     H
        POP     D
        POP     B
        POP     PSW
        RET
                                ;MULT11:  multiplies the value
                                ;..pointed to by DE by 11-byte
                                ;..value in the WORK area
MULT11:
        PUSH    B               ;save caller's loop control
        LXI     B,2             ;copy multiplier for shifting
        LXI     H,SHIFTER
        XCHG
        LDIR
        CALL    RIPLO           ;clear work area
        LXI     H,INDEX         ;copy multiplicand into WORK+2
        LXI     D,WORK+2
        LXI     B,9
        LDIR
```

(continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Four

```
        MVI     B,16            ;shift 16 bits
MULT01:
        LXI     H,SHIFTER+1
        ANA     A               ;clear carry
        RARR    M               ;test for add of partial product
        JZ      MSBIS0          ;..and possibly done
        DCX     H
        RARR    M
MULT03:
        JRC     INTERADD        ;add accum to partial result,
MULT04:                         ;..with more doubling to go
        PUSH    B
        MVI     B,11
        LXI     H,WORK+10
        ANA     A               ;clear carry
MULT05:
        RALR    M               ;double multiplicand by shifting
        DCX     H
        DJNZ    MULT05          ;..for all 11 bytes
        POP     B               ;restore loop control
        DJNZ    MULT01          ;do for all bits in SHIFTER
        POP     B               ;restore caller's loop control
        RET

INTERADD:
        PUSH    B
        LXI     B,11            ;add 11 bytes
        LXI     D,WORK+21       ;-> partial result
        LXI     H,WORK+10       ;-> doubled value
        ANA     A               ;clear carry
INTADD01:
        LDAX    D
        ADC     M
        STAX    D
        DCX     D
        DCX     H
        DJNZ    INTADD01
        POP     B
        DJNZ    MULT04          ;do for all bits in shifter
        POP     B
        RET

FINALADD:
        MVI     B,11
        LXI     D,WORK+21
        LXI     H,WORK+10
        ANA     A               ;clear carry
FLADD01:
        LDAX    D
        ADC     M
        STAX    D
```

```
        DCX     D
        DCX     H
        DJNZ    FLADD01
        POP     B
        RET

MSBIS0:                         ;arrive here if most signif byte
        INX     H               ;..of shifter is 0
        RARR    M               ;..and test for least signif 0
        JNZ     MULT03
        JRC     FINALADD
        POP     B
        RET

OFFSET  DS      1
SHIFTER DS      2               ;multiplier

        COMMON  /WORK/
WORK    DS      33
        COMMON  /RANDP/
RANDP   DS      256
        COMMON  /BITPERM/
BITPERM DS      256
        COMMON  /I6/
I6      DS      6
        COMMON  /INDEX/
INDEX   DS      9
        COMMON  /START0/
START0  DS      2
        COMMON  /STEP0/
STEP0   DS      2
        COMMON  /MOD0/
MOD0    DS      2

        END
```

End Listing Four

Listing Five

```
;-----------------------------------------------------------------;
;                                                                 ;
;  ENCIPHER:  Encipher one 32-byte block of text by               ;
;       substitution with the six rotors and permutation          ;
;       by BITPERM.                                               ;
;                                                                 ;
;       Macros          Zilog                                     ;
;       ======          =====                                     ;
;       SLAR r          SLA r                                     ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  ENCIPHER
        EXTRN   EXTRACT,SUBSF,PERMF,NXTBLK
```

(continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Five

```
ENCIPHER:
        PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H

        LXI     H,I6            ;address the vector which
                                ;..determines the order of
                                ;..the rotors
        MVI     B,6             ;loop control
ELOOP:
        MOV     D,M             ;set DE to relative page of
        MVI     E,0             ;..rotor tables
        INX     H               ;for next rotor
        PUSH    H               ;save I6 address
        LXI     H,ROTOR0        ;base of rotor tables
        DAD     D               ;HL -> rotor to be used
        PUSH    H               ;save rotor address
        LXI     H,START0        ;base of start addresses
        MOV     E,D             ;set DE to relative word of starts
        SLAR    E               ;E = E * 2
        MVI     D,0             ;now DE = index to correct start
        DAD     D               ;HL -> start value to be used
        MOV     E,M             ;get the start value to DE
        INX     H
        MOV     D,M
        POP     H               ;HL -> rotor
        CALL    EXTRACT         ;extract 256 bits from the rotor
        LXI     D,BLOCK
        CALL    SUBSF           ;perform the substitution
        MOV     A,B             ;permute after second & fourth
        CPI     5               ;..substitutions
        CZ      DOPERM
        CPI     3
        CZ      DOPERM
        POP     H               ;restore I6 pointer
        DJNZ    ELOOP           ;do for six rotors

        CALL    NXTBLK          ;advance the index and step rotors

        POP     H
        POP     D
        POP     B
        POP     PSW
        RET

DOPERM:                         ;do a permutation on block
        XCHG                    ;HL -> BLOCK
        LXI     D,BITPERM       ;address permutation table
        CALL    PERMF           ;forward permutation
        RET
```

```
        COMMON  /BLOCK/
BLOCK   DS      32
        COMMON  /I6/
I6      DS      6
        COMMON  /STARTO/
STARTO  DS      2
        COMMON  /ROTORO/
ROTORO  DS      256
        COMMON  /BITPERM/
BITPERM DS      256
        END
```

End Listing Five

Listing Six

```
;-----------------------------------------------------------------;
;                                                                 ;
;  DECIPHER:  Decipher one 32-byte block of text by               ;
;       substitution with the six rotors and permutation          ;
;       by BITPERM.                                               ;
;                                                                 ;
;       Macros          Zilog                                     ;
;       ======          =====                                     ;
;       SLAR r          SLA r                                     ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  DECIPHER
        EXTRN   EXTRACT,SUBSG,PERMG,NXTBLK

DECIPHER:
        PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H

        LXI     H,I6+5          ;address the vector which
                                ;..determines the order of
                                ;..the rotors
        MVI     B,6             ;loop control
DLOOP:
        MOV     D,M             ;set DE to relative page of
        MVI     E,0             ;..rotor tables
        DCX     H               ;for next rotor
        PUSH    H               ;save I6 address
        LXI     H,ROTORO        ;base of rotor tables
        DAD     D               ;HL -> rotor to be used
        PUSH    H               ;save rotor address
        LXI     H,STARTO        ;base of start addresses
        MOV     E,D             ;set DE to relative word of starts
        SLAR    E               ;E = E * 2
        MVI     D,0             ;now DE = index to correct start
        DAD     D               ;HL -> start value to be used
        MOV     E,M             ;get the start value to DE
        INX     H
        MOV     D,M
```

(continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Six

```
        POP     H               ;HL -> rotor
        CALL    EXTRACT         ;extract 256 bits from the rotor
        LXI     D,BLOCK
        CALL    SUBSG           ;perform the substitution
        MOV     A,B             ;permute after second and fourth
        CPI     5               ;..substitutions
        CZ      DOPERM
        CPI     3
        CZ      DOPERM
        POP     H               ;restore I6 pointer
        DJNZ    DLOOP           ;do for six rotors

        CALL    NXTBLK          ;advance the index and step rotors

        POP     H
        POP     D
        POP     B
        POP     PSW
        RET

DOPERM:
        XCHG                    ;HL -> BLOCK
        LXI     D,BITPERM       ;address permutation table
        CALL    PERMG
        RET

        COMMON  /BLOCK/
BLOCK   DS      32
        COMMON  /I6/
I6      DS      6
        COMMON  /STARTO/
STARTO  DS      2
        COMMON  /ROTORO/
ROTORO  DS      256
        COMMON  /BITPERM/
BITPERM DS      256
        END
```

End Listing Six

Listing Seven

```
;-----------------------------------------------------------------;
;                                                                 ;
;  SUBSF:  The forward substitution function.  Perform            ;
;       substitution on one 32-byte block of text by addition     ;
;       mod 2^256.                                                ;
;                On call, DE -> text block, on page boundary      ;
;                                in BLOCK common area,            ;
;                           HL -> key block, on page boundary in  ;
;                                WORK common area.                ;
;                On return, DE -> substituted text block          ;
;                All registers are saved.                         ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  SUBSF

SUBSF:
        PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H

        MOV     A,E             ;get DE pointing to last byte
        ADI     31              ;...can do it this way since
        MOV     E,A             ;...text is on page boundary
        MOV     A,L             ;do the same for key block
        ADI     31
        MOV     L,A

        MVI     B,32            ;add 32 bytes
        ANA     A               ;clear carry
FLOOP:
        LDAX    D               ;get byte of text
        ADC     M               ;add key to it
        STAX    D               ;save sum in text area
        DCX     H               ;decrement pointers
        DCX     D
        DJNZ    FLOOP

        POP     H
        POP     D
        POP     B
        POP     PSW
        RET

        END
```

End Listing Seven

(Continued on next page)

Listing Eight

```
;-----------------------------------------------------------------;
;                                                                 ;
;  SUBSG:  The inverse substitution function.  Perform            ;
;       substitution on one 32-byte block of text by              ;
;       subtraction mod 2^256.                                    ;
;               On call, DE -> text block, on page boundary       ;
;                               in BLOCK common area,             ;
;                        HL -> key block, on page boundary in     ;
;                               WORK common area.                 ;
;               On return, DE -> substituted text block.          ;
;               All registers are saved.                          ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  SUBSG

SUBSG:
        PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H

        MOV     A,E             ;get DE pointing to last byte
        ADI     31              ;...can do it this way since
        MOV     E,A             ;...text is on page boundary
        MOV     A,L             ;do the same for key block
        ADI     31
        MOV     L,A

        MVI     B,32            ;subtract 32 bytes
        ANA     A               ;clear carry
GLOOP:
        LDAX    D               ;get byte of text
        SBB     M               ;subtract key from it
        STAX    D               ;save sum in text area
        DCX     H               ;decrement pointers
        DCX     D
        DJNZ    GLOOP

        POP     H
        POP     D
        POP     B
        POP     PSW
        RET

        END
```

End Listing Eight

Listing Nine

```
;-----------------------------------------------------------------;
;                                                                 ;
;  PERMF:  permute bit vector BLOCK by BITPERM:                   ;
;          BLOCK <- BLOCK[BITPERM].                               ;
;                                                                 ;
;          On call, HL -> bit vector, BLOCK,                      ;
;                   DE -> permutation table, BITPERM              ;
;          Assumes that BLOCK, BITPERM and WORK are on page       ;
;          boundary.                                              ;
;          Saves all registers.                                   ;
;                                                                 ;
;        Macros          Zilog                                    ;
;        ======          =====                                    ;
;        JRNZ nn         JR NZ,nn                                 ;
;        SETB n,M        SET n,(HL)                               ;
;        BIT  n,M        BIT n,(HL)                               ;
;-----------------------------------------------------------------;

        MACLIB  Z80
        CSEG

        PUBLIC  PERMF
        EXTRN   RIPLO           ;routine to clear workspace

PERMF:
        PUSH    PSW             ;save registers
        PUSH    B
        PUSH    D
        PUSH    H

        CALL    RIPLO           ;clear partial result area
        LXI     B,0             ;initialize indexes

                                ;if Prth bit of BLOCK = 1 then
PRMF01:                         ;..Cth bit of WORK = 1
        LDAX    D               ;get BITPERM value
        RAR                     ;and convert to byte
        RAR                     ;..index into BLOCK (divide by 8)
        RAR
        ANI     1FH
        PUSH    H               ;save BLOCKS's base
        ADD     L
        MOV     L,A             ;HL -> byte containing desired bit
        PUSH    H               ;save BLOCK pointer
        LXI     H,EXBIT         ;-> second byte of BIT instruction
        MVI     A,46H           ;restore original BIT instruction
        MOV     M,A
        LDAX    D               ;get BITPERM value again
        ANI     07H             ;...and convert to bit index
        XRI     07H             ;reverses bit position in 0..7
        ADD     A               ;then shift to BIT's mask position
        ADD     A
        ADD     A
```

(continued on next page)

Listing Nine

```
        ORA     M
        MOV     M, A
        POP     H
        BIT     0,M
EXBIT   EQU     $-1
        JRNZ    PRMF20          ;if BITPERM[i] bit is on, set
                                ;..Cth bit of WORK
PRMF10:
        POP     H               ;restore BLOCK base
        INX     D               ;next BITPERM value
        INR     C               ;next bit of WORK
        DJNZ    PRMF01          ;B is counting down from 255
        LXI     D,WORK
        XCHG
        LXI     B,32
        LDIR
        POP     H
        POP     D
        POP     B
        POP     PSW
        RET
                                ;ith bit of BLOCK is on; set on
                                ;..Cth bit of WORK
PRMF20:
        MOV     A,C             ;find the jth byte of WORK
        RAR
        RAR
        RAR
        ANI     1FH
        LXI     H,WORK
        ADD     L
        MOV     L,A
        PUSH    H               ;save BLOCK pointer
        LXI     H,EXSET
        MVI     A, 0C6H         ;restore original SETB instruction
        MOV     M,A
        MOV     A,C             ;find the ith bit of the byte
        ANI     07H
        XRI     07H
        ADD     A
        ADD     A
        ADD     A
        ORA     M
        MOV     M,A
        POP     H
        SETB    0,M
EXSET   EQU     $-1
        JMP     PRMF10

        COMMON  /WORK/          ;the temporary workspace; this
WORK    DS      33              ;routine uses only 32 bytes,
        END                     ;..but others need 33
```

End Listing Nine

Listing Ten

```
;-----------------------------------------------------------------;
;                                                                 ;
;  PERMG:  Inversely permute bit vector BLOCK by                  ;
;          BITPERM: BLOCK <- BLOCK[1/BITPERM].                    ;
;                                                                 ;
;          On call, HL -> bit vector, BLOCK,                      ;
;                   DE -> permutation table, BITPERM.             ;
;          Saves all registers.                                   ;
;                                                                 ;
;       Macros          Zilog                                     ;
;       ======          =====                                     ;
;       JRC nn          JR C,nn                                   ;
;       SETB n,M        SET n,(HL)                                ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  PERMG
        EXTRN   RIPLO           ;routine to clear workspace

PERMG:
        PUSH    PSW             ;save all registers
        PUSH    B
        PUSH    D
        PUSH    H
        PUSH    H               ;save bit vector pointer

        CALL    RIPLO           ;clear partial result area
        XRA     A               ;get a zero
        STA     COUNT           ;zero bit count
        MVI     B,0             ;zero index
        MVI     C,32            ;outer loop control

PRMG01:
        MVI     B,8             ;inner loop control
        MOV     A,M             ;first byte of BLOCK
        PUSH    H               ;save BLOCK pointer
        LXI     H,COUNT         ;to count bits
PRMG02:
        ADD     A               ;test bit for 1
        JRC     PRMG20          ;if 1, set on BITPERM[i] bit
PRMG05:                         ;..of result
        INR     M               ;count the bit
        DJNZ    PRMG02          ;do for all bits
        POP     H               ;restore BLOCK pointer
        INR     L               ;next byte: BLOCK must be on page
        DCR     C               ;adjust outer loop
        JNZ     PRMG01          ;do for all bytes of BLOCK
        POP     H               ;return pointer to start of BLOCK
        LXI     D,WORK          ;replace input BLOCK with result
        XCHG
        MVI     C,32
```

(Continued on page 80)

File Encryption (Listing continued, text begins on page 44)

Listing Ten

```
        LDIR                    ;BLOCK <- BLOCK[1/BITPERM]
        POP     H
        POP     D
        POP     B
        POP     PSW
        RET

PRMG20:
        PUSH    PSW             ;save current byte in A
        MVI     A,0C6H          ;pristine 2nd byte of SET
        STA     EXSETG
        MOV     A,M             ;get bit count, ith bit
        PUSH    D               ;save BITPERM pointer
        MOV     E,A             ;works because BITPERM is on page
        LDAX    D               ;get BITPERM[i] byte
        RAR
        RAR
        RAR
        ANI     1FH
        PUSH    H               ;save bit counter pointer
        LXI     H,WORK          ;address partial result's ith byte
        MOV     L,A
        LDAX    D               ;get BITPERM[i] byte again
        ANI     07H             ;compute bit index into byte
        XRI     07H             ;...count bits from left to right
        ADD     A               ;then shift to SET's mask position
        ADD     A
        ADD     A
        PUSH    H               ;save partial result jth pointer
        LXI     H,EXSETG
        ORA     M
        MOV     M,A
        POP     H               ;restore partial result pointer
        SETB    0,M
EXSETG  EQU     $-1
        POP     H
        POP     D
        POP     PSW
        JMP     PRMG05

COUNT   DS      1               ;holds bit count

        COMMON  /WORK/
WORK    DS      33              ;the temporary workspace; this
                                ;..routine uses only 32 bytes,
                                ;but others need 33
        END
```

End Listing Ten

Listing Eleven

```
;-----------------------------------------------------------------;
;                                                                 ;
;  CYCLE:  Convert a permutation cycle to a permutation list      ;
;       at the nth permutation.                                   ;
;       BITPERM <- CYCLE(N, RANDP, BITPERM)                       ;
;                                                                 ;
;       On call:  DE -> BITPERM (start of 256 byte area on        ;
;                       page boundary).                           ;
;                 HL -> RANDP (256 bytes on page boundary).       ;
;                 A = N, 0 <= N <= 255.                           ;
;       On return:  BITPERM is established from RANDP and N.      ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  CYCLE

CYCLE:
        PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H
                                ;create BITPERM
        STA     OFFSET          ;save N
        LXI     B,0             ;loop control and ith counter
                                ;set DE to destination byte,
CYCLOOP:                        ;..DE+RANDP[C]
        MOV     L,C             ; -> RANDP[C]
        MOV     A,M
        MOV     E,A             ;DE -> BITPERM[RANDP[C]]
                                ;set HL to source byte
        LDA     OFFSET
        ADD     C               ;N+C
        MOV     L,A             ;HL -> RANDP[N+C]
        MOV     A,M
        MOV     L,A             ;HL -> RANDP[RANDP[N+C]]
        MOV     A,M             ;..get that byte and
        STAX    D               ;..put it in BITPERM[RANDP[C]]
        INR     C               ;next ith
        DJNZ    CYCLOOP         ;do for all bytes

        POP     H
        POP     D
        POP     B
        POP     PSW
        RET

OFFSET  DS      1

        END
```

End Listing Eleven

(Listing Twelve begins on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Twelve

```
;-----------------------------------------------------------------;
;                                                                 ;
;  UPTACK:  Base conversion.  ACCUM<-UPTACK(BASE,LENGTH,ARG)      ;
;    Convert digits represented by ARG to the base BASE.          ;
;    Output is in ACCUM (32 byte accumulator) in base 2.          ;
;    On Call: A = BASE: 1 < BASE < 256                            ;
;             B = LENGTH of ARG in bytes: 0 < LENGTH < 256        ;
;             HL -> ARG: byte string representing digits in       ;
;                   base BASE:  0 <= byte < BASE.                 ;
;    On return: HL -> base 2 representation of ARG, mod 2^256,    ;
;               in 32 byte accumulator                            ;
;                                                                 ;
;       Macros          Zilog                                     ;
;       ======          =====                                     ;
;       RARR M          RR (HL)                                   ;
;       JRC  nn         JR C,nn                                   ;
;       JRNC nn         JR NC,nn                                  ;
;-----------------------------------------------------------------;

        CSEG
        MACLIB  Z80

        PUBLIC  UPTACK
        EXTRN   RIPL0           ;routine to clear accumulator

UPTACK:
        PUSH    PSW
        PUSH    B
        PUSH    D

        STA     BASE            ;base used for shifting
        STA     BASER           ;..to refresh BASE
        SHLD    ARGP            ;save argument pointer
        PUSH    B               ;save loop control
        LXI     H,ACCUM         ;clear accumulator
        LXI     D,ACCUM+1
        LXI     B,31
        XRA     A
        MOV     M,A
        LDIR

        LHLD    ARGP            ;restore pointer
        POP     B               ;restore loop control
        JMP     UPTK02
UPTK01:
        CALL    MULT32          ;multiply accumulator by base
UPTK02:
        CALL    ADD32           ;add arg digit to accum
        INX     H               ;next arg digit
        DJNZ    UPTK01

        LXI     H,ACCUM         ;point to ACCUM on return
        POP     D
```

```
        POP     B
        POP     PSW
        RET

MULT32:
        PUSH    B               ;save loop control
        PUSH    H               ;save arg pointer
        CALL    RIPLO           ;clear work area

        MVI     B,8             ;shifter loop control
MULT01:
        LXI     H,BASE
        ANA     A               ;clear carry
        RARR    M               ;if least signif bit is on
        JRC     PARTIAL         ;..sum partial product
MULT02:
        PUSH    B
        MVI     B,32
        LXI     H,ACCUM+31
        ANA     A
MULT05:
        RALR    M               ;double multiplicand by shift
        DCX     H
        DJNZ    MULT05
        POP     B
        DJNZ    MULT01

        LXI     D,ACCUM         ;return product to ACCUM
        LXI     H,WORK
        LXI     B,32
        LDIR
        LDA     BASER           ;refresh base
        STA     BASE
        POP     H
        POP     B
        RET

PARTIAL:
        LXI     D,WORK+31       ;add ACCUM to WORK
        LXI     H,ACCUM+31
        PUSH    B
        MVI     B,32
        ANA     A
PART01:
        LDAX    D
        ADC     M
        STAX    D
        DCX     D
        DCX     H
        DJNZ    PART01
        POP     B
        JMP     MULT02

ADD32:
```

(Continued on next page)

File Encryption (Listing continued, text begins on page 44)

Listing Twelve

```
        LXI     D,ACCUM+31
        LDAX    D
        ADD     M
        STAX    D
        JRC     ADD05
        RET
ADD05:
        PUSH    B
        MVI     B,31
ADD07:
        DCX     D
        LDAX    D
        ADI     1
        STAX    D
        JRNC    ADD10
        DJNZ    ADD07
ADD10:
        POP     B
        RET

ARGP    DS      2               ;save pointer to argument
BASE    DS      1               ;hold base for shifting
BASER   DS      1               ;hold original base for refresh
ACCUM   DS      32              ;accumulator
        COMMON  /WORK/
WORK    DS      33              ;common work area

        END
```

End Listings

A Small-C Concordance Generator

by John Staneff

The program presented in this article builds a concordance of a source text file. Included in other source programs, it can become a high-level software tool. The utility extends an internal sort capability, which incorporates the use of word lists within the sort input phase.

What Is a Concordance?

The root word, *concord*, implies agreement. If a scholar studies the collected works of an author and finds that an excerpt from one portion is nearly identical to others, then he or she says that they are in concordance. If the passages are examined and key phrases listed alphabetically, referring back to their source, then this secondary work is called a concordance. In effect, the efforts of the scholar to document the agreement of the various passages takes on the name of that agreement.

An early illustration of a concordance is a scholar's examination of the Bible; another, an analysis of the complete works of Shakespeare. Today, any cross-reference is a proper subset of a concordance, whose full meaning is now much broader.

How Is a Concordance Used?

A concordance is used when you want to index something back to a source work or collection of works. It isn't a data base program, and it doesn't modify files or cause you to do so; it extracts the words you decide to extract (or excludes those you desire left out) and produces an index to their source.

One Example

While not referring specifically to a concordance generator and while reviewing another kind of software product entirely, Jerry Pournelle, in the April 1983 *BYTE* (see especially p. 348), described an excellent use for a concordance. In addition to being a columnist for *BYTE*, Pournelle is a well-known science fiction writer; he wanted to document his cast of characters, presumably for reuse in future writings. What Pournelle attempted to do was to extract his characters and to index them back to their source.

If he had had a concordance generator available, he could have located his characters by line number, page number, or paragraph (he even could have picked them up by chapter or book). If he wished, he could have used only those words that occurred in an inclusionary list, or he could have selected only those words that did not occur in an exclusionary list. He could have manually verified and edited the resultant data structure as required. Used in this way, the concordance becomes a high-level index to the original work.

Another Example

If a book publisher produces a quality text, the publisher usually provides an index to it. Traditionally, this requires that a highly skilled proofreader actually read the manuscript and draw out words that ought to be indexed. These words then are sorted, and the sorted list is reviewed for each word's worthiness for inclusion in the index. Finally, the index is prepared.

This process is unlikely to provide an accurate or adequate index. (Have you turned to a book's index only to find that either it doesn't exist or it's incorrect?) If the typesetting is redone as

1. Begin or set up the tree, which is re-executable during the program execution:

   ```
   cninit(list,type,dupl);
   ```

   where
   - 'list' is the character array containing the file name of the keyword list,
   - 'type' is the integer type of the list,
     - 0 indicates that lists aren't used,
     - 1 is exclusionary, and
     - 2 is inclusionary,
   - 'dupl' is the integer indicator for duplicates,
     - 0 is to keep all token-reference pairs, and
     - 1 is to exclude duplicates

2. Push data onto the tree, requiring both the key word and the reference point:

   ```
   cnbuild(linenumb,text);
   ```

   where
   - 'linenumb' is the integer reference, and
   - 'text' is the character array of text, each string is terminated by a null,

3. Order the tree, producing a sequence list of the sorted tree:

   ```
   cnorder( );
   ```

4. Pop data off the tree, returning the key word and associated reference pointer:

   ```
   i = cnxtrct(token);
   ```

   where
   - 'i' is the integer reference pointer, and
   - 'token' is the character array containing the key word returned (terminated by a null character).

Table

John E. Staneff, Jr., Rt. 1, Box 801, Ellensburg, WA 98926.

the work is revised, it is likely that the cross-references may not be corrected, or if corrected, not thoroughly.

If book indexing were automated, using inclusionary lists developed individually for several broad disciplines (such as medicine, computing, etc.), perhaps this ambiguity of publishing could be brought under control. Here a concordance generator could be used to great advantage.

Yet Another Example

How many students type their class notes? How many of them do so on their personal computer? And how many of them keep the files around after printing off the notes? If the notes were stored by lecture and indexed through a concordance generator, then the problem of how to study for an exam on a specific topic would be much simplified. (Just think of the advantage in an open book test!)

Going beyond the classroom, once the student becomes a doctor, lawyer, accountant, or computerist, wouldn't those indexed class notes be of interest in solving problems in professional practice? And why limit usage to scholastic topics? A recipe collection, treated as one large work, could surely be indexed as to all the various ways to fix hamburger.

A Concordance Is a Finding Tool

Think of the concordance routines as a high-level software tool. By themselves, they don't do much, but incorporated into a larger source program, they can perform significant tasks. The simplest way of using the routines is to treat them as an internal sort, keeping everything sent to them and returning it all at the end. (The only limit to this would be memory space.)

In such a simplistic view, each line of a poem might be sent separately, and the subsequent report would list only those lines occurring multiple times within the poem. Or perhaps a corporate branch office location table needs to be sorted. Because the routines are modular and includable, they are also reusable. The office phone book might be sorted by name, printed, and resorted by extension, all within the same program execution.

By increasing the level of sophistication of the calling module and electing to perform inclusionary or exclusionary checking, the routines can lead to much greater rewards. Perhaps the greatest reward comes when a series of steps culminates in a short list of particularly sensitive information drawn from a great amount of raw data. By cross-checking the index for the presence of two or more key words (or phrases) that refer to a single passage, one might quickly find answers to questions usually considered difficult to solve.

For instance, in what chapter of the Mabinogion does King Aurthur go hunting and is followed by Guenevere? If an index isn't handy, one might have to do a bit of reading! A more practical application might be to narrow down several symptoms to aid in illness diagnosis or to correlate law cases relating to a given set of circumstances. A program that combines lists of a general nature (words like *a*, *the*, *an*, etc.) to condense a work and build an index, then passes that result against words of a specific nature (words like *compiler*, *ALGOL*, etc.) to refine the subset, and finally correlates those results by reference pointer is a powerful finding tool.

Using Lists To Select Words

The concordance routine, at the programmer's option, can use an external word list to govern its word storage activity. Two different kinds of lists may be used: an inclusionary list and an exclusionary list.

In fact, the difference between the two kinds of lists rests in how we humans regard them. Any word list can be used as either inclusionary or exclusionary at the programmer's option (or user's, if the choice is selected by an

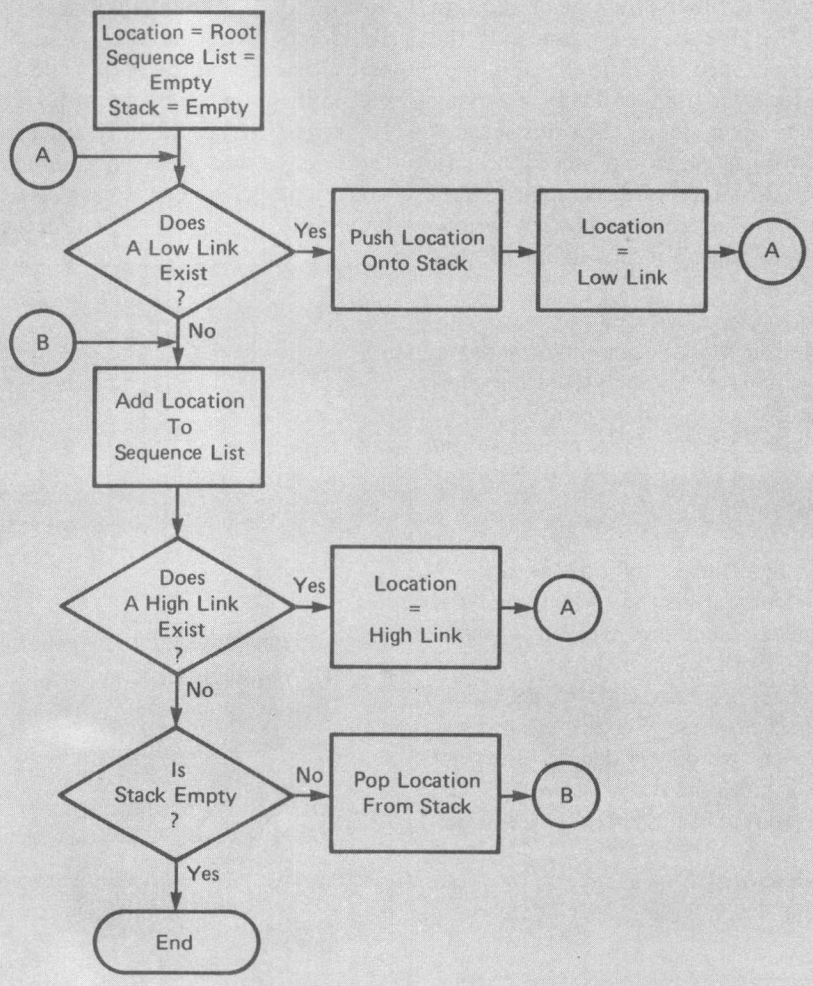


Figure 1
Procedure To Build Sequence List

externally supplied parameter).

The basic word list is simply constructed. Each entry is a word: an ASCII text string on a line by itself. The whole of the list is sorted alphabetically, and no duplicates are allowed. There are no special characters or hidden meanings and no trick start or end words; the list starts with the first word (on the first line) and ends with the end of file (no more lines in the list).

How the word list is used depends on the way it is described to the routines. If it is considered exclusionary, then only key words that do not appear in the list are accepted into the concordance routine's storage. If it is considered inclusionary, then only key words that appear in the list are accepted. In the end, only those words that have been accepted appear in the index.

By definition, if a data file is passed against an inclusionary list to build an index and that index is passed against the same list used as an exclusionary list, a net loss of the entire data file will occur with no resultant index. However, it may be desirable to filter data in several steps, using exclusionary lists, and then to draw upon inclusionary lists to extract a desired index. (Conceivably, separate inclusionary lists can be used to build several indexes, which can then be merged to provide an interdisciplinary index.)

This technique is especially useful when the initial data file is quite large and has an abundance of noise words (such as articles, adverbs, and so on). For the final pass, the indexer may select from a number of specially prepared and maintained "stock" word lists, each word list corresponding to a discipline's jargon or other criteria.

If multiple word lists are used to filter a source file, they may contain overlapping words. As long as the word lists are used consistently (i.e., both used as exclusionary lists), this will not present a problem. If they contain an overlap and aren't used consistently, the result is predictable: the overlapping words will *not* appear in the final result. One technique to "discover" overlapping words is to perform a simple cross-reference on one of the word lists while using the other word list as an inclusionary list. The result is a list of words present in both lists.

Certainly concordances may be prepared without using word lists, but employing the powerful utility that word lists can provide makes them invaluable tools.

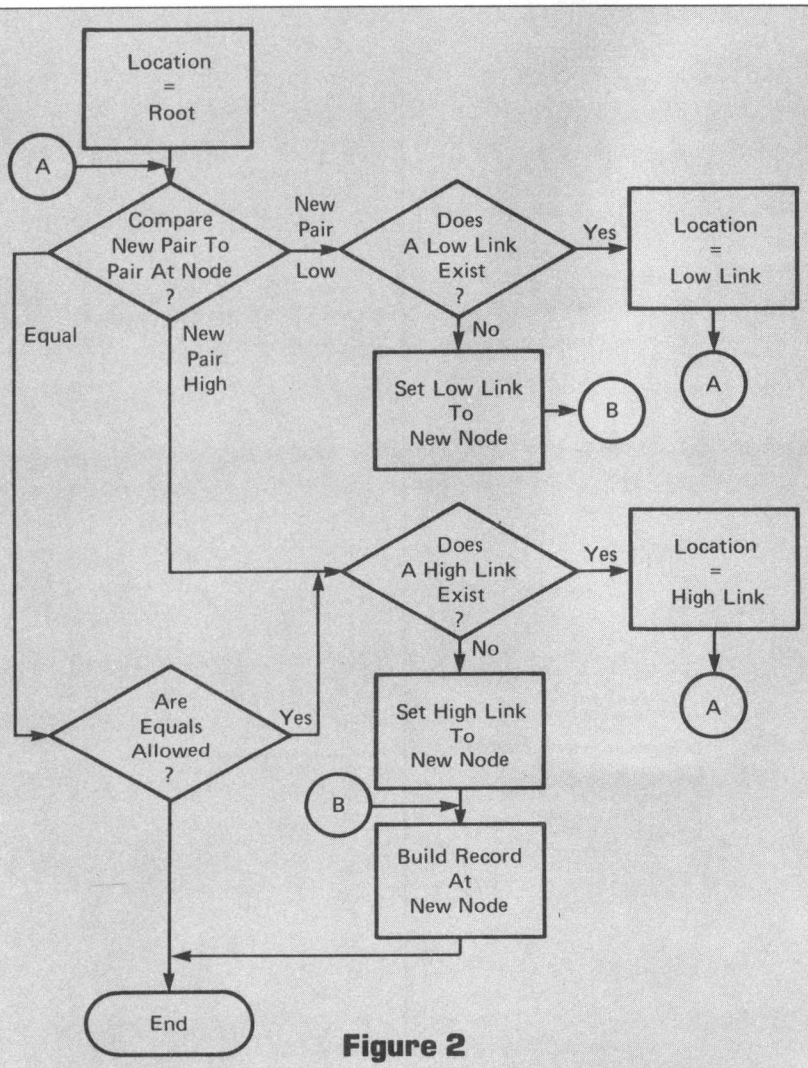


Figure 2
Procedure To Add To Tree

The Programs

Presented here are three modules written in Small-C and developed on the early CP/M version as supplied by The Code Works. They are CNDEF.C, CNCORD.C, and CONCRD.C (Listings One, Two, and Three, pages 108, 108, and 109, respectively).

The CNDEF.C module sets up default variables required by the CNCORD.C module.

The CNCORD.C module builds the actual concordance, accepting tokens given it by the mainline program and stuffing them into internal storage. A reference number is also filed and can be used as line number, page number, paragraph number, chapter number, or entire work number.

The CONCRD.C module is a mainline program that makes use of the concordance modules. It is intended for demonstration only, producing a simplified concordance of a short text file. CONCRD.C also uses the modules as the heart of a cross-reference program, the major difference being that a concordance program retains the source lines while a cross-reference program retains references to already printed lines. The four calls to CNCORD.C are described in the table (page 88).

The Algorithm

Internally, the CNCORD.C routine is a binary tree sort. It interleaves source data acquisition with sorting and, upon

comand, dumps the tree's contents back to the caller. This technique is fairly quick. Once the data is read (and listed, if desired), the retrieval of sorted tokens proceeds rapidly. The apparent speed is the result of overlapping I/O delays with CPU activity in building the tree.

The attached flow charts (Figures 1 and 2, pages 89 and 90) describe the tree building and retrieval processes. Entries are added to the tree by finding an appropriate location in which to add the new token-reference pair and "linking" it in through the low and high tree links. Duplicate pairs are added only if the initialization call has specified that duplicates are allowed; when duplicates are added, they are considered higher than current entries.

Finding the correct location in which to add a pair is a process of testing each node for its relative value to the new pair, taking the low link if the new pair is lower and the high link otherwise. When there is no link address to follow (because its value is zero), then that link address is made the new location, and the new pair is thereby linked.

Determining the order of the tree is an exercise in pushing and popping stacks. At each node, starting from the root (first in table), the node is tested for having a low link. If one is present, the node's address (table entry location) is pushed onto the stack, the low link is followed, and the process is repeated at the new node. If there isn't a low link, then the node's address is added to the sequence list. Next, if a high link is present, it is taken, with the process repeated (looking for a lower entry) from that node. If neither a high nor a low link is present, the stack is popped once, and the recovered address is added to the sequence list. The process continues from the point where it looks for a high link.

The process includes a sequence list, for the convenience of the output routine (cnxtrct(token)). When required, the programmer can restart the output by resetting the list pointer (cnthrdpt) to zero.

A special routine included in the CNDEF.C listing (usrvfy(token)) is used to arbitrarily exclude any key word that may be identified without needing to be in a list. The version supplied will exclude all one- and two-letter words and any "word" that contains a nonalphabetic character. A sample version to accept all key words presented to it is also provided in the comments of the supplied routine.

■■J

(Listings begin on page 90)

Sample Run

```
B†type verse.txt
Here with a loaf of bread beneath the bough,
A flask of wine, a book of verse—and thou
Beside me singing in the wilderness—
And wilderness is paradise enow.

"How sweet is mortal sovranty!"—think some:
Others—"How blest the paradise to come!"
Ah, take the cash in hand and waive the rest;
Oh, the brave music of a distant drum!

B†
```

The Input Text

```
B†type xcl.lst
ALL
AND
COME
FINI
HAND
HERE
OTHERS
OUR
REST
SOME
TAKE
THE
THERE
THRU
TILL
WHEN
WITH

B†
```

The List of Words to Exclude

```
B†concrd
Input file name: verse.txt
Exclusionary list name: xcl.lst

BENEATH
      Here with a loaf of bread beneath the bough,
BESIDE
      Beside me singing in the wilderness—
BLEST
      Others—"How blest the paradise to come!"
BOOK
      A flask of wine, a book of verse—and thou
BOUGH
      Here with a loaf of bread beneath the bough,
BRAVE
      Oh, the brave music of a distant drum!
BREAD
      Here with a loaf of bread beneath the bough,
CASH
      Ah, take the cash in hand and waive the rest;
DISTANT
      Oh, the brave music of a distant drum!
DRUM
      Oh, the brave music of a distant drum!

ENOW
      And wilderness is paradise enow.
FLASK
      A flask of wine, a book of verse—and thou
LOAF
      Here with a loaf of bread beneath the bough,
MORTAL
      "How sweet is mortal sovranty!—think some:
MUSIC
      Oh, the brave music of a distant drum!
PARADISE
      And wilderness is paradise enow.
      Others—"How blest the paradise to come!"
SINGING
      Beside me singing in the wilderness—
SOVRANTY
      "How sweet is mortal sovranty!"—think some:
SWEET
      "How sweet is mortal sovranty!"-think some:
THINK
      "How sweet is mortal sovranty!"—think some:
THOU
      A flask of wine, a book of verse—and thou
VERSE
      A flask of wine, a book of verse—and thou
WAIVE
      Ah, take the cash in hand and waive the rest:
WILDERNESS
      Beside me singing in the wilderness—
      And wilderness is paradise enow.
WINE
      A flask of wine, a book of verse—and thou
B†
```

The Output from CONCRD.C

Concordance (Text begins on page 86)

Listing One

```
/*
 * concordance global definitions and
 * variables.
 *
 * Includes the usrvfy() routine, which should be
 * custom modified by user to further analyze tokens before
 * they are filed into the sort tree.
 */

#define CNKEYSIZ        400     /* number of keywords possible */
#define CNKEYSPC        2500    /* amount of keyword space */
#define CNTKNLIM        1900    /* number of tokens that can be sorted */
#define CNTKNLM         1899    /* stack size, above minus one */
#define CNTKNSPC        12000   /* size of sort tree char data */
#define CNLNSZ          80      /* assume each line of 80 bytes */
#define CNDLSZ          30      /* size of delimiter list */

int cnexcl[CNKEYSIZ];   /* pointer list */
char cnextxt[CNKEYSPC]; /* Text pointed to */

int cnexhi;             /* Search - high */
int cnexlo;             /* Search - low */

int cnlsfl;             /* Keyword file number */
int cnlstp;             /* List type */
int cnsveqls;           /* 1 if save equals (token and reference), else 0 */

char cndelm[CNDLSZ];            /* Delimiter list */

char cntreedt[CNTKNSPC];        /* token sort tree storage area */
int cntreerf[CNTKNLIM]; /* storage for references */
int cntreelc[CNTKNLIM]; /* pointers to text data for tree entry */
int cntreehi[CNTKNLIM]; /* pointer to next high in tree */
int cntreelo[CNTKNLIM]; /* pointer to next low in tree (or equal) */
int cntreesk[CNTKNLM];  /* stack of return-to pointers */
int cnthread[CNTKNLIM]; /* pointer list showing sequence of sort-tree */
int cnbegin;            /* sort-tree indicator */
int cnpos;              /* current location in sort-tree */
int cnsub;              /* for threading tree */
int cnsubr;             /* current sort-tree subscript */
int cnstart;            /* where sort-tree begins */
int cnstack;            /* number items in return-to stack */
int cnthrdpt;           /* current location in sequence of sort-tree */


usrvfy(token)
char *token;
{

/*
 * User verification routine, should be in user's application
 * module (or in cndef.c).  In simplest form, is as follows:
 *
 *      usrvfy(token)
 *      char *token;
 *      {
 *              return(0);
 *              }
```

```
 *
 * This simple version includes all commers.
 *
 * The version supplied here excludes (returns a 1) if the
 * token is less than 3 characters long, or if all characters
 * in the token aren't in range 'A' to 'Z'.  Note that by
 * the time that tokens come here, they are all uppercase.
 */

int i;
char c;

        i=0;
        while ((c=token[i])!=0) {
                if (c<'A') return(1);
                if (c>'Z') return(1);
                i++;
                }
        if (i<=2) return(1);    /* keep if three characters or more */
        return(0);
        }

/* end of cndef.c */
```

End Listing One

Listing Two

```
 /*
  * Main code segment of the concordance routine.
  * Requires cndef.c or equivalent to define the
  * global variables and sizes.
  * User may wish to rewrite the usrvfy() routine,
  * and to extend delimiter list in cninit().
  */

 cninit(file,lsttype,savsw)
 char *file;
 int lsttype,savsw;
 {

 /*
  * Initialize routine for concordance.
  *
  * "file" is filename of keyword file
  * "lsttype" is a switch relating to the keyword file
  *       0 = not used
  *       1 = acts as an exclusionary list
  *       2 = acts as an inclusionary list
  * "savsw" is a switch relating to the need to retain duplicate references
  *       0 = keep all references
  *       1 = keep only one token per reference number
  *
  * Returns       0 if ok
  *               1 if lstfile not found (+ message on screen)
  */

 int flag, c, loc;

        flag = 0;       /* assume all ok */
        cnlstp = 0;     /* default is list not used */
        if (lsttype==1) cnlstp=1;       /* Exclusionary */
        if (lsttype==2) cnlstp=2;       /* Inclusionary */
        if (cnlstp!=0) {
```

(continued on next page)

Concordance (Listing continued, text begins on page 86)

Listing Two

```
        cnlsfl=fopen(file,"r");
        if (cnlsfl==0) {
                puts("Can't open keyword file");
                putchar(CR);
                flag=flag+1;
                }
        }

cnsveqls=0;     /* default is to keep everything */
if (savsw==1) cnsveqls=1;       /* except unless requested otherwise *.

/*
 * The delimiter list is a string of those characters
 * which act as token separators, such as spaces and
 * commas.
 */
cndelm[00]=' ';          /* space */
cndelm[01]='     ';      /* tab */
cndelm[02]=',';          /* comma */
cndelm[03]='.';          /* period */
cndelm[04]=';';          /* semi-colon */
cndelm[05]=':';          /* colon */
cndelm[06]='!';          /* exclamation point */
cndelm[07]='?';          /* question mark */
cndelm[08]='-';          /* dash */
cndelm[09]='%';          /* percent */
cndelm[10]='$';          /* dollar sign */
cndelm[11]='#';          /* pound sign */
cndelm[12]='*';          /* asterisk */
cndelm[13]='&';          /* ampersand */
cndelm[14]='(';          /* open parenthesis */
cndelm[15]=')';          /* close parenthesis */
cndelm[16]='@';          /* at sign */
cndelm[17]='=';          /* equals sign */
cndelm[18]='+';          /* plus sign */
cndelm[19]='~';          /* tilde */
cndelm[20]=0;            /* end of list */
/*
 * Note, the list might need to be extended, if so,
 * then also change the definition in cndef.c.
 */

/*
 * If the keyword list is available (file exists),
 * then read it into the stacks.  Note that the routine
 * assumes that the input file is ordered.  This would
 * require the user to pre-sort this file; not a bad
 * idea after any text-editing session.
 */

cnexlo=0;
cnexhi=0;
if (cnlstp!=0) {        /* list should be used */
        if (flag==0) {
                /* file is open... */
                loc=0;  /* main stacks subscript */
                cnexcl[++cnexhi]=loc;   /* starting loc of text */
                while ((c=getc(cnlsfl))>0) {
                        if (c!=CR) {
```

```
                                if ((c>='a') & (c<='z')) c=c-32;
                                /* make table all uppercase */
                                cnextxt[loc++]=c;
                                }
                        else {
                                cnextxt[loc++]=0;
                                cnexcl[++cnexhi]=loc;
                                }
                        }
                }
        /* at end of file, cnexhi is one higher than # keywords */
        }

/*
 * Initialize the sort-tree parameters too
 */

cnintree();

/*
 * End of Initialization


          */

         return(flag);
         }

 cnbuild(refr,text)
 int refr;
 char *text;
 {

 /*
  * Routine to extract each token from the line "text", and to include
  * it in the sort-tree depending on the presence of the token in the
  * keyword list, and the exclude/include philosophy selected by the
  * user.
  * The reference, "refr", is included along with the text as its
  * filed in the sort tree.  This variable can be what ever the user
  * decides it to be, for instance it can be a line number or a page
  * number.  In a more sophisticated approach, it can be used as a
  * paragraph number, chapter or verse number, or can be another type
  * of reference (ie: part of a linked list).
  */

 char token[30];
 int l,x,i,j,n,m;
 int p,q;
 char r;
        p=0;
        q=0;
        while ((r=text[p++])!=0) q++;
        l = p-1;        /* length of text/line including last null */

        x=1;
        while (1) {
                /* find and separate tokens */
                i=cntokn(x,text);       /* find next delimiter */
                if (i==0) i=l;  /* default to end of line if none found */
                /* move "text" from "x" to "i" to "token" */
                p=x-1;
                q=0;
                while (p<(i-1)) {
                        r=text[p++];
```

(Continued on next page)

Listing Two

```
                        token[q++]=r;
                        }
                token[q]=0;
                cnuprcse(token);        /* convert to uppercase */
                /*      */
                n=cnfind(token);        /* include/exclude test */
                if (n==0) {     /* not excluded */
                        m=usrvfy(token);
                        /* let application decide also */
                        if (m==0) {     /* still ok */
                                cnstuff(refr,token);
                                }
                        }
                x=i+1;
                if (x>=1) break;        /* continue to end of line */

                 }
        }


cntokn(cnstart,text)
int cnstart;
char *text;
{

/*
 * Returns the positional location of the next delimiter character
 * in the text string, from the starting position.  Returns zero
 * if none found.  All displacements are +1 relative to actual
 * position, including "cnstart" as supplied by caller.
 */

int i,j,k;
char c,s;

        i=0;
        while (i<(cnstart-1)) {
                /* find out if "cnstart" is past end of string */
                c=text[i++];
                if (c==0) return(0);    /* Yep, too short */
                }

        while ((c=text[i])!=0) {
                /* look at balance of string */
                j=0;
                while ((s=cndelm[j++])!=0) {
                        if (c==s) return(++i);  /* delimiter found */
                        }
                i++;
                }
        return(0);      /* no delimiter found */
        }
cnfind(token)
char *token;
{

/*
 * Searches the keyword list to find a match to the token
 * presented.
```

```
* Success or failure to match the token is translated to
* routine success or failure depending on "cnlstp".
* if cnlstp is 1, and token found, then exclude
*              1, and token not found, then include
*              2, and token found, then include
*              2, and token not found, then exclude
*
* Returns      1 to exclude
*              0 to include
*
* if "cnlstp" is 0, then always returns 0 (include)
*/

int n,h,l,m,p,c,found;
char test[30];

        if (cnlstp==0) return(0);

        found=0;        /* assume not found */

        /* following routine is a binary search */
        h=cnexhi;       /* range limits */
        l=cnexlo;
        while (1) {
                n=(h+l)/2;
                m=cnexcl[n];    /* current pointer */
                /* move in the pointed-to keyword */
                p=0;
                while ((c=cnextxt[m++])!=0) test[p++]=c;
                test[p]=0;
                /*      */
                m=cncmpr(token,test);   /* FORTRAN-like compare */
                if (m==0) {     /* found it */
                        found=1;
                        break;
                        }
                else if (m>0) l=n;
                else h=n;
                if ((h-l)<=1) {
                        found=0;        /* not in list */
                        break;
                        }
                }

        /*
         * Determine routine success based on following decision
         * table:
         *
         *    Keyword list not used          Y  N  N  N  N
         *    Exclusionary list              -  Y  Y  N  N
         *    Inclusionary list              -  -  -  Y  Y
         *    Token found in list            -  Y  N  Y  N
         *    ========================================
         *    Ok, return(0)                  X     X  X
         *    Fail, return(1)                   X        X
         */
        if ((cnlstp==1) & (found==1)) return(1);
        if ((cnlstp==2) & (found!=1)) return(1);
        return(0);
```

(Continued on next page)

Listing Two

```
        }

cncmpr(a,b)
char *a,*b;
{

/*
 * Performs a FORTRAN-like compare,
 * Returns -1 if "a" is less than "b"
 *          0 if "a" is equal to "b"
 *          1 if "a" is greater than "b"
 *
 * Comparison is of strings, from left to right, continuing
 * so long as both are still equal, and then stopping if
 * either are 0 (NULL)
 */
int r,s,t,q;
char c,d;

        r=0;
        q=1;
        s=0;
        t=0;
        while (q==1) {
                c=a[s++];
                d=b[t++];
                if (c<d) {
                        /* first is low */
                        r=-1;
                        q=0;
                        }
                else if (c>d) {
                        /* first is  high */
                        r=1;
                        q=0;
                        }
                else if (c==0) {        /* stop if end of string */
                        q=0;
                        }
                }
        return(r);
        }
cnuprcse(text)
char *text;
{

/*
 * Routine translates all lower case letters to their
 * upper case equivalents.
 */
int i;
char c;
        i=0;
        while ((c=text[i])!=0) {
                if ((c>='a') & (c<='z')) {      /* lower case found */
                        text[i]=c-32;           /* to upper */
                        }
                i++;
                }
        }
```

```
cnintree()
{
/* Initialize the sort-tree */

        cnbegin=1;
        cnpos=0;
        cnsub=1;        /* use 0 as null pointer, thus start at 1 */
        cnstart=1;
}
cnstuff(refr,token)
int refr;
char *token;
{

/* Add token and reference to sort-tree */

int l,tst,c,next,link;
int i;
char entry[30];

        if (cnbegin==1) {
                cnbegin=0;      /* turn off begin flag */
                cntree(refr,token);     /* add this token as first in tree */
                cnsub++;
                }
        else {
                cnsubr=cnstart;
                tst=1;
                while (tst==1) {
                        cngetree(entry,cntreelc[cnsubr]);
                        /* get entry at location of cnsubr */
                        c=cncmpr(token,entry);  /* FORTRAN-like compare */
                        if ((c==0) & (cnsveqls==1)) {
                                i=cntreerf[cnsubr];
                                if (i==refr) return;
                                }
                        if (c<0) {
                                next=cntreelo[cnsubr];
                                link=2;
                                }
                        /* Place equal on high tree for proper output */
                        /* listing with respect to reference numbers */
                        if (c>=0) {
                                next=cntreehi[cnsubr];
                                link=1;
                                }
                        if (next==0) {
                                tst=0;
                                }
                        else {
                                cnsubr=next;
                                }
                        }
                /* Thread in new entry when thread location found */
                cntree(refr,token);
                if (link==1) {
                        cntreehi[cnsubr]=cnsub;
                        }
                else {
```

(Continued on next page)

Listing Two

```
                        cntreelo[cnsubr]=cnsub;
                        }
                cnsub++;
                }
        }
cnorder()
{

/*
 * This routine determines the retrieval sequence (order) of the
 * sort-tree.  When the routine is complete, the array cnthread[]
 * is a pointer list to the sort-tree, in proper for a sequential
 * retrieval from the first entry.
 */

int tst;
int c;
int i;
char entry[30];
        cnthrdpt=1;     /* starting add-in entry for pointer list */
        cnstack=0;
        cnsubr=cnstart;
        tst=1;
        while (tst==1) {
                /* always look for a lower entry */
                c=cntreelo[cnsubr];
                if (c!=0) {
                        /* push stack when lower found */
                        cntreesk[++cnstack]=cnsubr;
                        cnsubr=c;
                        }
                else {
                        tst=cnpopstk();
                        }
                }
        /* whole tree now ordered into pointer list */
        cnthrdpt=0;     /* prime retrieval pointer */
        }

cnpopstk()
{

/* pop the top item in the tree cnstack */

char entry[30];
int c,t,tst;

        tst=1;
        t=1;
        while (tst==1) {
                /* no lower, print here */
                cngetree(entry,cntreelc[cnsubr]);
                cncrdout(cnsubr,entry);
                /* now look higher */
                c=cntreehi[cnsubr];
                if (c!=0) {
                        /* follows high link once, then lower */
                        cnsubr=cntreehi[cnsubr];
                        tst=0;  /* end but not end of cnstack */
                        }
                else {
```

```
                        /* pop stack when no higher found */
                        if (cnstack<=0) { /* are there any left in stack? */
                                t=0;
                                tst=0;  /* end AND end of stack */
                                }
                        else {
                                cnsubr=cntreesk[cnstack--]; /* pop again */
                                }
                        }
                }
        return(t);
        }

cntree(refr,token)
int refr;
char *token;
{
/* Add line and reference to tree */

int t;
        t=cnpush(token,cnpos);
        cntreerf[cnsub]=refr;
        cntreelc[cnsub]=cnpos;
        cntreehi[cnsub]=0;
        cntreelo[cnsub]=0;
        cnpos=t;
        }
cnpush(a,b)
char *a;
int b;
{
/* Moves contents of a into the position referenced by b in cntreedt */

int t,q,s;
char c;

        s=0;
        t=b;
        q=1;
        while (q==1) {
                c=a[s++];
                cntreedt[t++]=c;
                if (c==0) q=0;
                }
        return(t);
        }
cngetree(a,b)
char *a;
int b;
{
/* moves the content of cntreedt[b] to a[?] until end of string */

int t,q,s;
char c;
        s=0;
        t=b;
        q=1;
        while (q==1) {
                c=cntreedt[t++];
                a[s++]=c;
                if (c==0) q=0;
                }
        }
cncrdout(subr,entry)
```

(Continued on next page)

Concordance (Listing continued, text begins on page 86)

Listing Two

```
int subr;
char *entry;
{

/*
 * Add current subscript pointer (cnsubr) to a-building ordered
 * pointer list.  Assume that the variable cnthrdpt is pointing
 * to the new location in the pointer list.  (All new locations
 * are set to zero so that in the case of end of list, it is
 * properly identified.
 */

        cnthread[cnthrdpt++]=subr;
        cnthread[cnthrdpt]=0;
        }

cnxtrct(token)
char *token;
{

/*
 * Function to return the sort-tree in order as determined by cnorder().
 * The global variable cnthrdpt, set to 0 by cnorder(), is assumed to
 * be the location of the previously listed token.
 * The variable "token" is set by the function when there
 * is data to return.
 * Function returns     0 if end of list detected, no data values returned
 *                      else, returns the particular reference number
 */

int i;
int n;

        i=cnthread[++cnthrdpt];
        if (i!=0) {
                n=cntreelc[i];  /* get location of text for this subscript */
                i=cntreerf[i];  /* return i as the reference */
                cngetree(token,n);      /* get text */
                 }
         return(i);
         }
 /* end of cncord.c */
```

End Listing Two

Listing Three

```
/*
 * CONCRD.C
 *
 * This program presents a simplified concordance of an input file.
 *
 */

#define CR      13      /* carriage return */

char    text[8000];     /* text storage area */
char    filnm[50];      /* name of input file */
char    tokn[80];       /* returned token */
```

```
char    ptokn[80];      /* prior returned token */
char    line[80];       /* single text line */

int     lines[200];     /* line pointers to text array */
int     numlines;       /* number of lines stored */
int     flvbl;          /* variable associated to file */
int     chrloc;         /* current byte on text[] */


main()          /* init, load, sort, and list concordance */
{
int lsttyp,savsw;
        numlines=0;     /* no lines yet stored */
        chrloc=0;       /* no characters in text storage yet */
        opnfile();      /* open input file */
        lsttyp=1;       /* Exclusionary List */
        savsw=1;        /* Save token only once per reference */
        puts("Exclusionary list name: "); gets(tokn);
        cninit(tokn,lsttyp,savsw);      /* init concordance routine */
        loadtxt();      /* build concordance input arrays */
        cnorder();      /* determine sequence */
        prtcncrd();     /* display concordance */
        fclose(flvbl);  /* close file */
        }

opnfile()       /* open input file */
{
        while (1) {
                puts("Input file name: "); gets(filnm);
                if ((flvbl=fopen(filnm,"r"))!=0) break; /* get file? */
                puts("Can't open file"); putchar(CR);
                }
        }

loadtxt()       /* load text from input file into text array */
{
int i;
        while ((i=getln(line,flvbl))>0) {       /* while data */
                lines[++numlines]=chrloc;       /* count lines */
                mvlnto();       /* save in text array */
                cnbuild(numlines,line);         /* build concordance */
                }
        }

prtcncrd()      /* print concordance */

{
int i;
        ptokn[0]=0;     /* null token */
        while ((i=cnxtrct(tokn))!=0) {  /* while tokens */
                mvlnfro(i);     /* get line that token points to */
                if ((i=cncmpr(tokn,ptokn))!=0) {        /* borrow routine */
                        puts(" "); putchar(CR);
                        puts(tokn); putchar(CR);
                        svtkn();        /* save as new prior token */
                        }
                puts("     "); puts(line); putchar(CR); /* display line */
                }
        }
```

(Continued on next page)

Concordance (Listing continued, text begins on page

Listing Three

```
mvlnto()        /* move contents of line to text at chrloc onward */
{
int i,j;
        j=0;
        while ((i=line[j++])!=0) text[chrloc++]=i;
        text[chrloc++]=0;
        }

mvlnfro(i)      /* move line by number from text to line */
int i;
{
int j,k,l;
        l=0;
        j=lines[i];
        while ((k=text[j++])!=0) line[l++]=k;
        line[l]=0;
        }

svtkn()         /* save current token in previous token */
{
int i,j,k;
        k=0; j=0;
        while ((i=tokn[j++])!=0) ptokn[k++]=i;
        ptokn[k]=0;
        }

getln(input,file)       /* get line of input from file */
char *input;
int *file;
{
int c,i;
        i=0;
        while ((c=getc(file))>0) {
                input[i++]=c;
                if (c==CR) { input[i]=0; return(1); }   /* <cr> ? */
                }
        line[i]=0;
        return(c);
        }

/* end of concrd.c */

#include a:cndef.c
#include a:cncord.c
```

End Listings

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Son of Floating-Point Benchmark

In case there are any newcomers out there, perhaps it is appropriate to supply some background to this month's column. We first published BASIC and PL-I versions of Bill Savage's floating-point speed and accuracy benchmark program in the September 1983 *DDJ* The subject seemed to catch the fancy of many readers, and versions of the benchmark were subsequently contributed and published for Pascal and Forth (November 1983), Fortran (January 1984), C (March 1984), Logo and LISP (June 1984), and 8086/8087 assembly language (July 1984). A reprise of the BASIC benchmark, a Modula-2 listing, T/Maker III, and a corrected Logo version accompany this month's column (Listings One–Four, pages 108–109).

A preliminary collection of benchmark results appeared in the March 1984 *DDJ* issue, and a greatly expanded table of results accompanies this month's column (page 111). We have timings for computers ranging from the mighty Cray X-MP to the humble Sinclair! We now have a very good representation of the programming languages, except for the little-known and arcane language known to its initiates as COBOL (apparently COBOL programmers read *Datamation* instead of *DDJ*.

As in the March table, the column FPP contains the type of hardware floating-point support that was used, if any was noted by the contributor. Most of the timings were accurate only to the nearest second (except for the very fast minicomputers and mainframes); they are displayed to three decimal places as an artifact of the dBASE II report program. The Error column was calculated as ABS-(2500.00000-A), converting to scientific notation and rounding the mantissa to the nearest integer. Obviously, the smallest errors correspond to the most accurate results.

There are some pretty interesting figures buried in the table. Digital Research Personal BASIC and Dimension 68000's BASIC lead the microcomputer field for grossness in absolute error, while IBM BASIC has the same distinction in the mainframe world. On the other hand, Digital Research Logo turned in an absolutely correct result; a classic example of one hand not knowing what the other hand is doing. Computer Innovations' new "Optimizing" C86 version 2 turned in a slower time than its previous, presumably "Non-Optimizing," C86 version 1.3. Morgan Computing's Professional BASIC interpreter for the IBM PC cranked out a timing and absolute error result competitive with the best compilers, while simply blowing the doors off every other known microcomputer interpreter.

Along with the timings and error results, I had the good fortune to receive many interesting and educational letters, some of which are excerpted below.

Benchmarking Spreadsheets

Thomas W. Moran of Bethesda, MD, writes: "I tried three spreadsheets (a perverse misapplication, I know!). Multiplan running on a Burroughs B20 (5 mHz 8086, I believe) took 24 minutes and gave an answer of perfect accuracy. T/Maker III, running on a 4 mHz Z80, took 31 minutes and gave 2500.00027709 as the answer. Running on a Nippon Univac system (5 mHz 8088, I think), T/Maker III gave the same answer but took only 27 minutes. Lotus 1-2-3 running on an IBM PC could only calculate up to 2048, not 2500 (at least without using a more complex program). I estimate its 2500 iteration time at about 8 minutes. It also gave a perfectly correct answer.

"Regarding accuracy: by adding an instruction to the T/Maker III program to request rounding to 9999.999999 after each iteration, I forced T/Maker to give a perfectly correct answer. I suspect that both Multiplan and Lotus automatically round to integer or a few decimal places after each iteration, thus preventing the error from growing. I don't have convenient access to either, or I would have tried a starting value of, say, SQRT (2), thus forcing non-integer calculations. A final subtraction of SQRT (2) – 1.0 would then give results comparable to the other benchmark runs. Lotus was so much faster that I suspect it uses binary floating-point arithmetic. T/Maker uses decimal; I presume so does Multiplan. Does anyone know for sure?"

Benchmarking Modula-2

As everyone who reads Jerry Pournelle's *BYTE* column knows, Jerry's son Alex has proclaimed Modula-2 to be the language that will cure all the world's ills Real Soon Now. Chris Dunford writes:

"The attached listing is a Modula-2 version of the [Savage] benchmark, written for the M2M-PC compiler from the Modula Research Institute in Provo, Utah.

"The performance of the compiler in this particular test is far from impressive. Using single-precision reals in software (no 8087) on an IBM PC, the test took about 2 minutes and 38 seconds; the final value of A was 2262.9. A long time for such a bad result.

"The redeeming quality of the compiler is its price: $40 plus a couple of bucks for shipping. The Institute is a nonprofit organization dedicated to spreading the gospel of Modula-2; the M2M compiler and its associated utilities were ported directly (in M-code) from the Lilith computer, Wirth's dedicated Modula machine. It appears that only the M-code interpreter and some low-level stuff were actually reworked for the PC.

"Although the system does not appear to be adequate for commercial

value, and each deviation is normalized so a true relative error is accumulated. Finally, the RMS error provides the average error for calculations over the range tested. Overall, it presents a more interpretable view of the errors in any given language's floating-point package.

"In the next issue of the newsletter from Digital Acoustics, another point is brought up. The arc tangent removes most of the error generated up to this step in the calculation. This is the result of having to map a number in the range from negative infinity to positive infinity into a range from –pi/2 to pi/2. Since most of the numbers in the benchmark have arc tangents that are very close to pi/2, any errors that have been picked up along the way are virtually removed. The error that is left comes from the error intrinsic to the arc tangent function .... The next step, the job of the tangent function, is to take an arc in the range –pi/2 to pi/2 and map it onto the range negative infinity to positive infinity. This is fraught with trouble, especially when all our test numbers are very nearly pi/2. The small error introduced by the arc tangent function becomes magnified by the tangent function. It turns out that the error introduced in this step is pi/2*N*error.

"To be redundant, let's write this another way. Let the test number be N, the correct result of the arc tangent be A, and the error introduced by the arc tangent function be 'err.' Then, the error after the tangent function will be

Tan(A+err)/N =
relative error = err * pi/2

"The relative RMS error calculated by the Peterson algorithm is about 2267 times the relative RMS error plus the relative RMS errors of the rest of the functions. These latter errors will be almost entirely masked by the magnified arc tangent error. Therefore, dividing the relative RMS error by 2267 provides an estimate for the number of accurate bits in the floating-point package, unless the package has a particularly bad arc tangent function.

"In summary, the error in the arc tangent function dominates the errors accumulated by both the Savage and the Peterson algorithms by virtue of the magnification induced by the tangent function. The Peterson algorithm provides a way to estimate the accuracy of the floating-point package by making use of this dominance, a nice bonus."

■■J

16-Bit (Text begins on page 106)

Listing One

Bill Savage's floating-point speed and accuracy benchmark in BASIC.

```
100 ' time and general accuracy test program
110 defint i
120 iloop=2500
130 a=1
140 for i=1 to iloop-1
150 a=tan(atn(exp(log(sqr(a*a))))) +1
160 next i
170 print using "a=####.####";a
180 stop
```

End Listing One

Listing Two

Modula-2 version of the Savage benchmark, contributed by Chris Dunford.

```
(* DDJ Floating Point Test for M2M-PC Modula-2 Compiler *)

(*$R-,T-*)  (* turn off run-time checks *)

MODULE fptest;

FROM MathLib0  IMPORT sqrt, exp, ln, arctan, sin, cos;
FROM RealInOut IMPORT WriteReal;
FROM Terminal  IMPORT WriteString, WriteLn;
```

```
CONST
  MaxLoop = 2500 ;

VAR
  a, x: REAL;
  i: CARDINAL;

BEGIN
  a := 1.0;
  FOR i := 1 TO MaxLoop-1 DO
    x := arctan(exp(ln(sqrt(a*a))));
    a := sin(x) / cos(x) + 1.0     (* no TAN function *)
    END;
  WriteString("A = "); WriteReal(a, 12);  WriteLn
END fptest.
```

End Listing Two

Listing Three

T/Maker III code for Savage floating-point benchmark, contributed by Thomas W. Moran.

```
ex   9999.999999
uc1  +*sqr
uc2  +log+exp
uc3  +atn+tan
uc4+ 1
uc5  pas
+    1
ex   ,
+
     ...                        2500 '+' lines
+
ex   9999.99999999
+
+
+
+
+
```

End Listing Three

Listing Four

Corrected Logo version of the Savage benchmark. This listing runs on *"DR. LOGO"*; the names of the transcendental functions may be slightly different for other implementations.

```
to savage
make "a 1
repeat 2499 [make "a (tan arctan exp log sqrt :a * :a) + 1 ]
print [a = ] :a
end
```

End Listings

FLOATING POINT BENCHMARKS FOR DDJ

COMPUTER	MHZ	LANGUAGE	VERS	FPP	TIME (SEC)	ERROR	SOURCE	COMMENTS
CRAY X-MP		CFT FORTRAN	X.11		0.000	2E-10	MARK SEAGER	ACTUAL TIME 0.00012 SEC
CRAY-1S		CFT FORTRAN	1.11		0.003	3E-5	MARK SEAGER	
CDC 7600		FORTRAN	FTN5		0.069	9E-6	MARK SEAGER	
CDC CYBER 170-875		BASIC			0.103	7E-6	DAVID SACHS	
CDC CYBER 170-875		FORTRAN			0.144	3E-20	DAVID SACHS	DOUBLE PREC.
CDC CYBER 170-875		FORTRAN			0.470	9E-6	DAVID SACHS	SINGLE PREC.
HONEYWELL		MULTICS FORTRAN			0.500	4E+1	SHERMAN GROMME	SINGLE PREC, TIME APPROX.
IBM 370		WATERLOO BASIC			0.570	2E-1	KARL CASPER	
DEC VAX 11/780		VMS FORTRAN-77			0.578	<1E-3	SHERMAN GROMME	SINGLE PREC
HP 1000-F		FORTRAN 4	2040		0.630	2E+2	COMPUSERVE	
IBM 3081		PL/I			0.660	2E-25	COMPUSERVE	
DEC 2060		TOPS-20 FORTRAN 4	V6		0.770	7E-1	P. O'KANE	SINGLE PREC.
DEC 20 KL-10		TOPS-20 FORTRAN	V5A		0.830	1E+1	LEHMAN M. JONES	CPU TIME, SINGLE PREC.
HONEYWELL		MULTICS FORTRAN			1.000	1E-3	SHERMAN GROMME	DOUBLE PREC, TIME APPROX
DEC VAX 11-780		FORTRAN 77			1.000	7E-10	HORST SALZWEDEL	
UNIVAC 1100/81		FORTRAN	TFOR 24J		1.020	2E-9	R. SCHLAIFER	DOUBLE PREC.
DEC VAX 11/780		VMS FORTRAN-77			1.054	<1E-3	SHERMAN GROMME	DOUBLE PRECISION
UNIVAC 1100/81		FORTRAN	FTN 10R1		1.120	2E-9	R. SCHLAIFER	DOUBLE PREC.
DEC 2060		TOPS-20 BASIC+2			1.300	4E+1	P. O'KANE	SINGLE PREC.
DEC 2060		TOPS-20 FORTRAN 4	V6		1.500	1E+0	P. O'KANE	DOUBLE PREC.
IBM 370		IBM BASIC			1.530	2E+3	KARL CASPER	PRESUMED SINGLE PREC
DEC 20 KL-10		TOPS-20 FORTRAN	V5A		1.870	8E-3	LEHMAN M. JONES	CPU TIME, DOUBLE PREC.
HP A700		FORTRAN 4			2.000	3E-8	COMPUSERVE	
DEC VAX 11-780		C			2.000	7E-10	HORST SALZWEDEL	
IBM PC (8088)	4.77	ASSEMBLER		8087	2.200	3E-10	CHRIS DUNFORD	
IBM PC (8088)	4.77	WL SYSTEMS FORTH		8087	3.200	2E-13	JOHN GOTWALS	
IPL 4441		PASCAL/VS			3.470	<1E-4	JEFF FURGAL	EQUIV. IBM 4343/2
8086	5.0	PL/I-86	1.01	8087	3.700	2E+1	BILL SAVAGE	MICROFLOAT LIBRARY
HP 9000 (68000)		FORTRAN (UNIX)			4.000	2E-6	COMPUSERVE	
HP 1000-F		FORTRAN FTN4	2040		4.000	5E-3	COMPUSERVE	
8086	5.0	PL/I-86	1.01	8087	4.000	2E+1	COMPUSERVE	
HP 9000		FORTRAN			4.200	<1E-6	UNKNOWN	
IBM PC (8088)	4.77	MVP-FORTH		8087	4.300	<1E-3	C. SPRINGER	
DEC PDP 11/44		RSX-11M FORTRAN		FIS	4.500	2E+2	JOHN TOSCANO	
DEC VAX 11/780		PASCAL	2.1		5.000	6E-10	COMPUSERVE	
GRID COMPASS (8086)		MS FORTRAN 77	3.1	8087	5.000	1E-9	HORST SALZWEDEL	
IBM PC (8088)	4.77	QNX C		8087	5.100	<1E-8	MICRO BUS. APPL	
NEC APC (8086)		MS FORTRAN 77		8087	5.100	<1E-2	K. FERGUSON	
IBM PC (8088)	5.0	QNX C		8087	5.300	1E-9	HORST SALZWEDEL	
IBM PC (8088)	5.0	MS FORTRAN 77	3.1	8087	5.800	1E-9	HORST SALZWEDEL	
IBM PC (8088)	4.77	MICROSOFT PASCAL	3.11	8087	6.000	<1E-3	J. SPEISER	DBL PREC, SEATTLE 87.LIB
IBM PC (8088)	4.77	MICROSOFT FORTRAN	3.1	8087	6.200	<1E-3	J. SPEISER	DBL PREC, SEATTLE 87.LIB
DEC PDP 11/44		RSX-11M FORTRAN		FIS	6.900	2E-1	JOHN TOSCANO	DOUBLE PREC
IBM PC (8088)	4.77	CI C86	2.10C	8087	6.920	2E-9	RICHARD LARSON	DOUBLE PREC.
DEC PDP 11/34		RT-11 FORTRAN		FIS	7.500		J. SPEISER	
IBM PC (8088)	4.77	BASIC COMPILER	1.00	8087	7.800	1E-3	J. SPEISER	DBL PREC, SEATTLE 87.LIB
HP 1000F		BASIC			8.000	3E+2	COMPUSERVE	

Table I

FLOATING POINT BENCHMARKS FOR DDJ

COMPUTER	MHZ	LANGUAGE	VERS	FPP	TIME (SEC)	ERROR	SOURCE	COMMENTS
COMPUPRO 8088	5.0	DESMET C	2.2	8087	8.000	5E+0	R. WEITZMAN	DOUBLE PREC.
COMPUPRO 8088	5.0	SUPERSOFT FORTRAN	1.04	8087	10.000	2E+2	R. WEITZMAN	SINGLE PREC.
8085	5.0	RMAC		8232	10.200	5E-3	BILL SAVAGE	MICROFLOAT LIBRARY
8085	5.0	PL/I-80	1.40	8232	10.400	5E-3	BILL SAVAGE	MICROFLOAT LIBRARY
8085	5.0	BASIC-80	5.20	8232	10.700	<1E-3	BILL SAVAGE	MICROFLOAT LIBRARY
NAT'L SEMIC. 16000	6.0	UNIX C	4.1B		12.000	<1E-4		DOUBLE PREC.
8085	5.0	FORTRAN-80	3.40	8232	12.500	5E-3	BILL SAVAGE	MICROFLOAT LIBRARY
COMPUPRO 8085	6.0	FORTRAN 80	3.3	9511	14.000	2E+2	R. WEITZMAN	SINGLE PREC.
IBM PC (8088)	4.77	NC P-SYSTEM	C1F	8087	14.000	3E-10	CHRIS DUNFORD	COMPILED TO NATIVE CODE
IBM PC (8088)	4.77	BASIC COMPILER	1.00	8087	14.200	<1E-3	J. SPEISER	DBL PREC, MICROWARE LIB
IBM PC (8088)	4.77	MORGAN PROF. BASIC	1.00	8087	15.200	<1E-8	RAY DUNCAN	
6502	4.0	UCSD P SYSTEM	II	8231	15.500	2E+2	STEVEN SPEARS	
8088	4.7	UCSD PASCAL	IV.1	8087	18.000	2E-7	COMPUSERVE	
LMC 16032	6.0	C			18.000	<1E-4	R. SCHLAIFER	
APPLE II (6502)	1.75	MVP-FORTH		9511	18.500	2E+2	C. SPRINGER	
LMC 16032	6	UNIX FORTRAN 77			21.000	<1E-5	R. SCHAIFER	
DEC PDP 11/44		RSX-11M BASIC-PLUS-2			21.800	1E+2	JOHN TOSCANO	SINGLE PREC
DEC MINC 11/23		FORTRAN		FPU	22.000	2E+2	BILL SAVAGE	
DEC LSI 11/23		FORTRAN			25.000	2E+2	K. FERGUSON	SINGLE PREC.
DEC PDP 11/44		RSX-11M BASIC			27.200	1E+2	JOHN TOSCANO	
HEATH H-89 (Z-80)	4.0	FORTRAN-80		9511	31.000	2E+2	JOHN TOSCANO	
DEC PDP 11/44		RSX-11M FORTRAN			34.600	1E+2	JOHN TOSCANO	
DEC MINC 11/23		MINC BASIC	2.0	FPU	38.000	2E+2	BILL SAVAGE	
HP 9186		HP BASIC			44.000	3E-7	COMPUSERVE	
HP 9836 (68000)		BASIC			44.290	<1E-3	BILL SAVAGE	
IBM PC (8088)	4.77	BASIC COMPILER	1.00		48.000	9E+1	J. SPEISER	SINGLE PREC
SAGE (68000)	8.0	P-SYSTEM PASCAL	4.12		54.000	3E+2	JOE BARNHART	SINGLE PRECISION
DEC LSI-11/23		C (UNIX)	V7		56.000	6E-8	COMPUSERVE	
GRID COMPASS (8086)		GRIDBASIC	3.0	8087	56.000	1E-9	HORST SALZWEDEL	
SAGE (68000)	8.0	P-SYSTEM FORTRAN	4.12		56.000	3E+2	JOE BARNHART	SINGLE PRECISION
SAGE (68000)	8.0	P-SYSTEM BASIC	4.12		58.000	3E+2	JOE BARNHART	SINGLE PRECISION
DEC LSI 11/23		FORTRAN			59.000	<1E-2	K. FERGUSON	DOUBLE PREC.
NEC APC (8086)		CI C86		8087	65.000	<1E-2	K. FERGUSON	
DEC LSI-11/23		PASCAL-2	2.1A	FP	66.000	1E-7	COMPUSERVE	
WANG PC (8086)	8.0	BASIC INTERP	1.0.2		68.000	2E+2	RAY DUNCAN	
IBM PC (8088)	4.77	SUPERSOFT FORTRAN	1.07		70.000	9E+1	HUGH KAWABATA	SINGLE PREC
DIMENSION (68000)	8.0	BASIC			76.000	1E+3	JOE BARNHART	40 X 24 SCREEN
SUN WORKSTN (68000)		UNIX CC COMPILER			85.000	<1E-3	JOHN TOSCANO	
IBM PC (8088)	4.77	CI C86	1.330	8087	85.000	1E-3	J. SPEISER	DOUBLE PREC
8086	5.0	BASIC-86	5.20		92.200	3E+2	BILL SAVAGE	
DEC PDP 11/44		RSX11M FORTRAN			103.000	2E-1	JOHN TOSCANO	DOUBLE PREC
DIMENSION (68000)	8.0	BASIC			103.000	1E+3	JOE BARNHART	80 X 24 SCREEN
DEC PDP 11/44		RSX-11M BASIC-PLUS-2			113.600	<1E-3	JOHN TOSCANO	DOUBLE PREC
68000	8.0	UCSD PASCAL	IV.12		115.000	3E-7	COMPUSERVE	
IBM PC (8088)	4.77	IBM APL			117.000	1E-8	J. SPEISER	
HP 9825T		HPL			117.000	7E-4	WAYNE BORGLUM	
6809	2.0	PASCAL			119.000	8E+0	ANDY BALL	MICROWARE SYSTEMS CORP
8088	8.0	DIGITAL RESEARCH C			119.000	<1E-4	GUY SCHARF	
HP 1000E		FORTRAN 4	2026		121.000	3E-8	COMPUSERVE	

Table I

(Continued on next page)

FLOATING POINT BENCHMARKS FOR DDJ

COMPUTER	MHZ	LANGUAGE	VERS	FPP	TIME (SEC)	ERROR	SOURCE	COMMENTS
8085	5.0	FORTRAN-80	3.40		140.800	2E+2	BILL SAVAGE	
8085	5.0	BASIC-80 COMPILER	5.20		140.800	2E+2	BILL SAVAGE	
HP 9835B		BASIC			140.800	1E-3	J. SPEISER	
ZENITH Z-100 (8088)		ZBASIC			142.730	2E+2	BILL SAVAGE	
6809	2.0	BASIC09			149.000	2E-2	ANDY BALL	MICROWARE SYSTEMS CORP
8088	8.0	SSS FORTRAN	1.04		154.000	2E-7	COMPUSERVE	
IBM PC (8088)	4.77	BASIC INTERPRETER			157.000	3E+2	J. SPEISER	SINGLE PREC
IBM PC (8088)	4.77	M2M-PC MODULA 2			158.000	2E+2	CHRIS DUNFORD	
IBM PC (8088)	4.77	BASIC COMPILER	1.00		170.000	1E-3	J. SPEISER	DOUBLE PREC
8086	8.0	MS PASCAL	?		171.000	1E-7	COMPUSERVE	
VAX 11/750		FRANZ LISP			172.000	6E-8	MICHAL YOUNG	4 USERS, TIME APPROX.
8085	5.0	BASIC-80	5.20		174.900	2E+2	BILL SAVAGE	
8086	5.0	PL/I-86	1.01		179.600	8E+2	BILL SAVAGE	
Z-80	4.0	BASIC-80	5.30		184.000	2E+2	COMPUSERVE	
HEATH H-89 (Z-80)	4.0	MBASIC COMPILER	5.2		189.700	2E+2	JOHN TOSCANO	
HEURIKON (68000)	8.0	UNIX SYSTEM III C			199.300	<1E-4	BOB HALLORAN	DOUBLE PREC.
HEATH H-89 (Z-80)	4.0	FORTRAN-80			203.000	2E+2	JOHN TOSCANO	
TANDY 16B (68000)		XENIX CC COMPILER			211.000	1E-3	JOHN TOSCANO	
IBM PC (8088)	4.77	SUPERSOFT FORTRAN	1.07		211.000	<1E-3	HUGH KAWABATA	DOUBLE PREC
68000	6.0	C (TRS-XENIX)			212.000	1E-6	COMPUSERVE	
IBM PC (8088)	4.77	DRI PERSONAL BASIC	1.0		217.000	1E+3	RAY DUNCAN	SINGLE PREC
HEATH H-89 (Z-80)	4.0	MBASIC	4.8		229.800	2E+2	JOHN TOSCANO	
SAGE (68000)	8.0	P-SYSTEM PASCAL	4.12		234.000	1E-4	JOE BARNHART	DOUBLE PRECISION
SAGE (68000)	8.0	P-SYSTEM FORTRAN	4.12		236.000	1E-4	JOE BARNHART	DOUBLE PRECISION
ZENITH Z-100 (Z-80)		MBASIC	5.22		236.740	2E+2	BILL SAVAGE	
8085	6.0	C/80	3.0		238.000	3E+2	COMPUSERVE	
SAGE (68000)	8.0	P-SYSTEM BASIC	4.12		238.000	1E-4	JOE BARNHART	DOUBLE PRECISION
DEC PDP 8		OS/8 FORTRAN IV			240.000	1E+3	SHERMAN GROMME	SINGLE PREC, TIME APPROX
8085	5.0	PL/I-80	1.30		254.400	8E+2	BILL SAVAGE	
IBM PC (8088)	4.77	DRI PERSONAL BASIC	1.0		255.000	1E+3	RAY DUNCAN	DOUBLE PREC
68000	6.0	BASIC/S-16	1.7		266.000	2E+2	COMPUSERVE	
VECT.GR. 4 (Z-80)		BASIC INTERPRETER	5.213		269.000	2E+2	RONALD WAGNER	DOUBLE PRECISION
HP 85		HPL			282.000	6E-4	WAYNE BORGLUM	
HP 86-A		BASIC			286.000	1E-3	MARK BAILEY	NATIVE MODE
8088	8.0	COMP. INNOV. C86	1.33		288.000	7E-6	COMPUSERVE	
HEATH H-89 (Z-80)	2.0	BHBASIC			310.000	2E+1	PEARSON/CAIRO	
BBC ACORN (6502)	2.0	BBC BASIC	V2		311.280	7E-1	P. O'KANE	SINGLE PREC.
HP-75 CALCULATOR		BASIC			325.000	6E-4	COMPUSERVE	
Z-80	4.0	BASIC-E	2.2		330.000	5E+1	COMPUSERVE	
NEC 8201		MS BASIC			363.000	2E+2	HORST SALZWEDEL	
HEATH H-89 (Z-80)	4.0	C/80	3.0		370.700	3E+2	JOHN TOSCANO	
Z-80	4.0	PL/I-80	1.3		373.000	9E+2	COMPUSERVE	
OSBORNE I (Z-80)		C/80			420.000	2E+2	BOB BRIGGS	
TANDY I (Z-80)		FORTRAN-80			422.000	2E+2	KARL CASPER	SINGLE PREC
APPLE II (6502)	1.75	APPLESOFT			463.000	<1E-3	C. SPRINGER	
APPLE II & 8088	1.75	APPLESOFT BASIC			470.000	<1E-3	R. S. SCHLAIFER	
Z-80	4.0	NEVADA FORTRAN	2.2		473.000	8E+1	COMPUSERVE	
VIC-20 (6502)	1.0	BASIC	2.0		477.000	9E-5	COMPUSERVE	
IBM PC	4.77	LOTUS 1-2-3			480.000	<1E-3	THOMAS MORAN	2048 ITER. TIME APPROX.

Table I

(Continued on page 114)

FLOATING POINT BENCHMARKS FOR DDJ

COMPUTER	MHZ	LANGUAGE	VERS	FPP	TIME (SEC)	ERROR	SOURCE	COMMENTS
TANDY I (Z-80)		MOLINERX PASCAL	5.1		485.000	1E+2	KARL CASPER	
TANDY I (Z-80)		BASIC			512.000	1E+2	KARL CASPER	
COMMODORE 64 (6510)		BASIC			514.000	<1E-3	TERRY THOMAS	
COMMODORE		PETSPEED	2.6		515.400	9E-5	DTACK GROUNDED	
NEC APC (8086)		CI C86			524.000	<1E-2	K. FERGUSON	
KAYPRO II (Z-80)	2.5	TURBO PASCAL			538.300	1E+0	MICHAL YOUNG	
Z-80	4.0	ZBAS	1.4		540.000	2E+2	COMPUSERVE	
IBM PC (8088)	4.77	TURBO PASCAL	1.00		544.000	5E-3	JEFF FURGAL	TAN FUNCTION SUPPLIED
TANDY CC (6809)		FORTH			560.000	<1E-3	GARY BERGSTROM	
TANDY CC (6809)		BASIC RS-CC	1.0		585.000	<1E-3	GARY BERGSTROM	
HEATH H-89 (Z-80)	2.0	MICROSOFT BASIC			620.000	2E+1	PEARSON/CAIRO	
APPLE MACINTOSH		MS BASIC	1.0		643.000	8E-6	HORST SALZWEDEL	DOUBLE PREC.
ZENITH ZW110 8088	5.0	CI C86			655.000	<1E-3	A. F. HERBST	
IBM PC (8088)	4.77	CI C86	1.33D		695.000	1E-3	J. SPEISER	DOUBLE PREC
IBM PC (8088)	4.77	DR. LOGO	1.0		750.000	2E-7	RAY DUNCAN	
IBM PC (8088)	4.77	CI C86	2.10C		764.670	7E-6	RICHARD LARSON	DOUBLE PREC.
TANDY 16B (68000)		XENIX MBASIC			773.600	1E-3	JOHN TOSCANO	
IBM PC (8088)	4.77	CI OPTIMIZING C86	2.05C		774.000	<1E-4	JEFF FURGAL	DOUBLE PREC.
GRID COMPASS (8086)		DESMET C	2.2		796.000	4E+0	HORST SALZWEDEL	
IBM PC (8088)	4.77	BASIC INTERPRETER	2.0		890.000	1E-3	J. SPEISER	DOUBLE PREC
CROMEMCO Z-80	4.0	32K STRUCTURED BASIC	3.66		900.000	3E-3	A. ROXBURGH	
CROMEMCO Z-80	4.0	16K BASIC	5.70		900.000	3E-3	A. ROXBURGH	
Z-80	2.0	NEVADA FORTRAN	3.0		975.000	8E+1	SHERMAN GROMME	
SINCLAIR ZX-81	3.5	BASIC			990.000	3E-1	PEARSON/CAIRO	"FAST" MODE
8085	6.0	CB-80	1.2		997.000	2E+1	COMPUSERVE	
IBM PC (8088)	4.77	DESMET C	2.2		1149.000	4E+0	JEFF FURGAL	DOUBLE PREC.
8085	6.0	FORTRAN-80	3.4		1251.000	1E-12	COMPUSERVE	
BURROUGHS B20 (8086)	5.0	MS MULTIPLAN			1440.000	<1E-3	THOMAS MORAN	
NIPPON UNIVAC (8088)	5.0	T/MAKER III			1620.000	<1E-3	THOMAS MORAN	
8085	6.0	CBASIC	2.06		1623.000	2E+1	COMPUSERVE	
ZILOG Z-80	4.0	T/MAKER III			1860.000	<1E-3	THOMAS MORAN	
Z-80	4.0	BASIC-80	5.3		1980.000	1E-7	COMPUSERVE	
Z-80	4.0	AZTEC C II	1.05		2190.000	1E-5	COMPUSERVE	
TI 99-4A (990X)		BASIC			2295.000	1E-3	PEARSON/CAIRO	
TRS 100 (80C85)	2.4	MS BASIC			2400.000	<1E-3	ALAN STEIN	
EAGLE II		CBASIC			2640.000	1E+1	DTACK GROUNDED	
TI-59 CALCULATOR					3240.000	7E-3	COMPUSERVE	
HP 41C					3500.000	<1E-2	K. FERGUSON	
Z-80	2.5	S-BASIC			3900.000	3E+4	COMPUSERVE	
HP 41CV CALCULATOR					3920.000	1E-2	RONALD WAGNER	
TANDY I (Z-80)		FORTRAN-80			4602.000	<1E-3	KARL CASPER	DOUBLE PREC
TANDY I (Z-80)		PASCAL-80			5220.000	<1E-3	KARL CASPER	
Z-80	4.0	JRT PASCAL	2.1		5760.000	1E-3	COMPUSERVE	
SINCLAIR ZX-81		BASIC			5760.000	3E-1	PEARSON/CAIRO	"SLOW" MODE
HP-65 CALCULATOR					6000.000	9E-1	COMPUSERVE	
HP 15C CALCULATOR					10200.000	1E-2	KARL CASPER	

End Table

Table I

C/UNIX PROGRAMMER'S NOTEBOOK

By Anthony Skjellum

In this column, we'll discuss some of the feedback received in response to previous columns, as well as some miscellaneous remarks.

uucp (Unix to Unix Copy)

Mike Meyers of Norman, OK, suggested that I publish my uucp address. Via this address, users on other Unix systems can send me mail electronically by using the standard Unix mail command. A uucp address is specified relative to a well-known site. Thus, the address "ucbvax|cithep|tony" should allow most users to reach me. If your site cannot communicate with ucbvax (UC Berkeley) directly, more site names must be prepended onto this address and onto those listed below as well. On the other hand, if your site can communicate with cithep (Caltech High Energy Physics) directly, you may omit the ucbvax prefix.

If a continuing electronic dialog develops, I will occasionally publish the uucp addresses of those involved so that others may join in. At the moment, the following users are participating in a discussion about Unix:

ucbvax|mtxinu|ea|jab
(Jeff A. Bowles)
ucbvax|cithep|yekta
(Yekta Gursel)
ucbvax|mtxinu|ea|emjej
(James Jones)
ucbvax|mtxinu|ea|mwm
(Mike Meyers)

I will include interesting remarks from this discussion in future columns for the benefit of those readers who cannot access the electronic mail.

C Users Group

Phillip K. Landis of Satellite Beach, FL, inquired about the address of the C Users Group of Yates Center, KS, which I mentioned in the April column. Their address and telephone are:

C Users Group
103 E. Rutledge
Yates Center, KS 66783
(316) 625-3554

This particular group has 34 volumes of C programs and offers them in a variety of disk formats (only for microcomputers). I would like to print the addresses of any other public domain C repositories. If you know of one, please forward me their address.

Public Domain C

Robert E. Fiorini of Albany, NY, writes: "I'm a new C user trying to find an inexpensive (possibly public domain) C compiler (with source code . . . if possible). My own efforts have been fruitless. I'm hoping that you or one of your readers may know where I can find such a compiler. If you can help me in any way . . . please HELP !! . . . It's funny, the only thing holding me back from the world of C is the C compiler itself."

Rod Cain's Small C is available through the C Users Group listed above. An enhanced version (not public domain) for 8080/Z80 CP/M systems is available (with source code) from The Code Works of Goleta, CA. (You can find their address in any number of *DDJ* back issues.) This particular compiler (Q/C) is a serious subset implementation and includes full compiler source for $95.00. An IBM version of this product should be available later this year and would be well worth the investment if priced comparably to the CP/M version. For $25 you can get version 2.1 of the Small-C compiler from J. E. Hendrix at Box 8378, University, MS 38677-8378 (see page 60 of the May 1984 *DDJ* for details). An MSDOS (IBM PC) version of Small C v.2.0 is available for $35 from The Coriolis Company, Box 76, Clinton Corners, NY 12514.

Comments about C Input-Output

Mike Meyers writes: "What you haven't seemed to realize is that almost every flaw in Unix also appears in C. C is terse, doesn't protect the user, and is poorly documented. The only documentation for C is K&R [Kernighan and Ritchie], which may be well written but is vague and inconsistent on all the points you turn to when you start implementing the language on new machines. To make matters worse, nobody (and I do mean nobody) sells a compiler that conforms to K&R, not even AT&T. I don't think anybody ever has, in any case. AT&T distributes a version of pcc [portable C compiler] that met K&R internally, but I think that by the time it was released externally, C had grown past K&R."

Gerald I. Evenden of N. Falmouth, MA, responded about C input-output as follows, with quite a different point of view:

"I was very disturbed with a basic concept about the C programming language that you kept implying in your column in the April issue. First of all I suggest that you carefully read the beginning paragraph of chapter 7 of Kernighan and Ritchie's book: *The C Programming Language*. It begins with 'Input and Output are not part of the C Language . . . .' What remains is a description of I/O procedures contained in a standard Unix library, which will take care of most filter types of functional operations. I have generally found them to be quite adequate for most programming efforts involving stream data and simple question-answer types of console I/O."

I would like to state that I am fully aware of the distinction between the C language definition and the standard input-output library. When teaching students how to program in C, this is

one of the first points I emphasize: C is a language that shows no special favoritism to a specific set of input-output routines. (A single standard set does exist, and this is the Unix standard.) I consider input-output independence an essential feature of C, but I feel that a discussion of a real C compiler environment cannot always be separated from a discussion of the support library that comes with it. I also maintain that the Unix input-output library is more than adequate for dealing with stream operations. Mr. Evenden summarizes the distinction between C and C input-output as follows:

"The beauty of C is that it doesn't have a plethora of specialized built-in functions but rather provides the programmer with a rich facility to build tools required for his own, occasionally specialized, needs. Obviously, we shouldn't have to redesign all the wheels needed, so most suppliers of C compilers include a library of functions patterned after the Unix libraries. But remember, there is absolutely no requirement to use them if they don't fit your needs, and they should only be viewed as a preliminary tool kit."

One point that merits further exploration is that of portability. While it is well and good to preach the separation of C and C input-output, only software that uses standard Unix input-output calls (and routines built on them) has a prayer of being moved readily between different machines or even between different compilers on the same machines.

Mr. Evenden continues: "I suspect that your problem with 'getc' et al. is related to screen editing and control, which is a category of program that doesn't fall into the 'filter' class of functions emphasized by Unix (and its libraries), and I certainly agree that these functions don't work in this case .... The astute programmer writes 'rawin( )' and 'rawout( )' to satisfy those needs. There's nothing to prevent it and everything to encourage it .... The worst possible outcome of the problems posed in your article is to even remotely suggest rewriting the current stream I/O functions. Their current form is a de facto standard, and a consistency of implementation is expected by most C programmers."

I really don't expect anyone to throw away the existing stream functions. Not only would this be unreasonable, it would also be undesirable. However, I do feel that clean raw input-output should be a supported capability; it needn't be reinvented each time a Unix programmer discovers that stream input-output is inconvenient for interactive purposes. In discussing a similar worry expressed by Mr. Meyers, I summarized my argument by stating that interactive programs comprise a large fraction of those run by Unix users, and that when programmers write interactive programs, they don't usually want the users treated like input files.

Despite Mr. Evenden's outspoken letter, we actually agree in many respects. However, some things he brought up demand careful examination. He states:

"The principal point of this complaint is that you should be a little more careful of what you are talking about. Writing about problems with C I/O is impossible since C I/O doesn't exist. However, your less-experienced readers will take your complaint to heart and decide that C is a useless language because Mr. Skjellum, et al., don't like the optional I/O library supplied with their compiler. If you were more positive in your approach you would be telling readers how to write their own procedures to do specialized console I/O on Unix, CP/M, etc. I've done it on both CP/M and Unix and found it to be a piece of cake in both cases and never gave the 'getc' group a second thought when it was obvious that they were not meant for the job at hand."

The comments I have made are based on several years of experience with C under Unix, CP/M, VMS, etc. One must come to grips with reality. C input-output, while optional, is normally what users must utilize to deal with a problem at hand; consequently, inexperienced users must lean more heavily on the standard library than experienced users. It is meaningful to discuss C input-output. It does exist, and Mr. Evenden discusses several points about it before stating that the topic is beyond the realm of discussion. Inexperienced users learn by reading dialog between others who have seen problems in their own work. Censoring this information to "protect" such users from disenchantment with C is an unacceptable alternative.

We have not reached the computer millenium. C and Unix as existing tools have flaws and drawbacks. Only through discussion can we seek solutions and create better future systems. The idea of restricting discussions because of semantic points seems to be contrary to that goal. Some other writers have taken the approach that C and Unix are wonderful tools and heap praise on them in review after review. A certain group of individuals feel highly insulted if this approach is not followed. In an evolving field, it makes sense to criticize as part of the learning process. That is why I include Mr. Evenden's final remarks, because I think that he has drawn a counterproductive conclusion from an understandable point of view:

"C is not a perfect language but it certainly beats what's in second place. Consequently, my enthusiasm about C makes me very chauvinistic about misplaced and invalid criticisms. I have a couple of minor complaints about some aspects of C but I bite my tongue when I think about the dark ages. After several happy years with ALGOL in the 60s I was sentenced to over 10 long years of Fortran purgatory before being born again with C. I guard this language jealously and you had better be careful of what you write or I'll curse you to a task of debugging 10,000 lines of BASIC code."

I hope that Mr. Evenden will reconsider and send in his comments about C so we can add them to the discussion.

BDS C Runtime Solution

Alex Cameron of Malvern (Victoria), Australia, had the following comments: "I couldn't help responding to your notes on the nonstandard nature of some of BDS C's runtime routines (*Dr. Dobb's* No. 90). There is probably little doubt that most of us gladly suffer its irregularities because of its speed, low price, and because it is arguably one of the finest C compilers around—all this notwithstanding, I still find the nonstandard buffered file functions such as fopen the most frustrating, simply because of the need to

continually declare buffers."

The listing on page 120 (listing stdlib3.c) is Mr. Cameron's proposed solution to this problem under BDS C.

Unix Comments

Two users had responses to previous comments about Unix. Tim Prince of Marblehead, MA, writes:

"As a new Unix user I find some of the subjects raised in your column this month [April] interesting. Certainly, in comparison with VAX/VMS, there is a lack of discipline in adhering to uniform standards for a user interface. This is the almost inevitable result of the way Unix evolved as the product of the work of various organizations. Also the volume of official documentation is much less, but if you count the many good books available from third parties, the comparison will soon go the other way. . . . Unix has collected a set of useful information into a pair of volumes, which the individual can afford to own."

He adds: "I have spent nearly 20 years learning things about GCOS, which has all the faults of which Unix is accused without many of the advantages. This operating system will soon disappear from the face of the earth, making our knowledge of its quirks so much nonbiodegradable mental trash. What is learned about Unix is more likely to remain useful after the current hardware is gone."

Mike Meyers writes: " . . . I tend to agree with most of the things you've said about Unix. The documentation is, at best, terse. What's worse, it's either missing or wrong in many places. To top it off, most of it assumes that you have access to the source and are a good computer scientist. To make matters still worse, there really isn't an online documentation system. What you have is an interactive manual printer. It is very definitely *not* friendly to the *naive* user. The best description I have run across is that Unix is expert-friendly. If you know it, it's going to be great. If you don't know it, well, that's just too bad."

These two viewpoints are not mutually exclusive. Unix is expert-friendly and beginner-hostile; many of the Unix functions such as man and mail are primitive, but this bothers some users less than others.

One can argue that, like C input-output, specific Unix tools are optional and can be replaced by programs preferable to the user. By this reasoning, it is not meaningful to complain about specific Unix features/programs. However, I feel that most users want to take Unix facilities off the shelf and spend their time on an application, not on reworking existing software tools. Therefore, comments about deficiencies in specific Unix tools are also germane.

Conclusion

This column exists for the purpose of discussing C and Unix as they are, with the problems they have. As stated emphatically above, C and C input-output are independent concepts. However, users generally deal with the input-output library, and deviations from that library have given users cause for complaint. Therefore, such a discussion has a place here. I invite readers to submit any code that illustrates how to deal with raw input-output in C under CP/M, Unix, or other operating systems.

In this column, we have considered some user comments about C/Unix based on several letters received recently. I would be interested to hear what others think about this ongoing discussion, as well as follow-up comments from those readers represented here.

■■J

C/UNIX Listing (Text begins on page 116)

```
/*
** stdlib3.c -- standard I/O library
**
** Copyright 1984  A. Cameron
*/
#include "bdscio.h"
/*
**
**      Standard fopen
**
**           return fd on success, NULL on error
**
*/

sfopen(filename,mode)
char *filename;
char *mode;
{
        int fd;

        if (*mode == 'w')       /* write mode */
        {
                if (!(fd = alloc(BUFSIZ)))
                        return(NULL);
                else
                {
                        if (fcreat(filename,fd) == -1)
                        {
                                free(fd);
                                return(NULL);
                        }

                        return(fd);
                }

        }

        if (*mode == 'r')       /* read mode */
        {
                if (!(fd = alloc(BUFSIZ)))
                        return(NULL);
                else
                {
                        if (fopen(filename,fd) == -1)
                        {
                                free(fd);
                                return(NULL);
                        }


                        return(fd);
                }
        }
```

```
        if (*mode == 'a')       /* append mode */
        {
                if (!(fd = alloc(BUFSIZ)))
                        return(NULL);
                else
                {
                        if (fappend(filename,fd) == -1)
                        {
                                free(fd);
                                return(NULL);
                        }

                        return(fd);

        }

        return(NULL);   /* failure */

}

/*
**
**      Standard fclose
**
*/

sfclose(fd)
int fd;
{
        fputc(CPMEOF,fd);

        if (!(fclose(fd)))
        {
                free(fd);
                return(NULL);
        }

        return(ERROR);
}
```

End Listing

PC-Write

Company:
Quicksoft, 219 First N.
#224, Seattle, WA 98109
Computer: IBM PC
Price: $10; Full registration $75

Reviewed by Ronald G. Parsons

The number of good programs available for the IBM PC under marketing arrangements similar to Freeware is increasing. This is a good sign for both users and the industry as a whole. Users are able to obtain good, if not the best, software at a reasonable price, and the small software developers are able to gain some market share without requiring a million dollar advertising campaign to get started. PC-Write is a good example of this trend.

Bob Wallace, PC-Write's designer, is making it available under a variation of Freeware he calls commission Shareware. If you register your copy of PC-Write with Quicksoft and someone obtains a copy from you and registers it, Quicksoft will give you a commission of $25, one-third of the registration fee. Register your copy, give away three copies, and, when they are registered, you will have all the advantages of registration at no cost. Registration has several benefits: telephone support, a bound copy of the documentation, and the complete source code (Pascal and assembler).

PC-Write provides a full set of word processing functions without a lot of the frills that may or may not be useful to many users. All the functions necessary for editing letters and short documents are provided. However, a way to merge several files together is only weakly provided, and there is no spelling checker. I consider these two features to be extremely valuable—almost mandatory. The program does create standard PC-DOS files so that many standard spelling checkers will work, including IBM's Word Proof, which I consider the best value available today in PC software.

Printing functions are provided with a separate program so that true "what you see is what you get" is not available. Support is provided only for dumb printers, although users can imbed printer control escape sequences and control codes if they desire; that does not come under the heading of ease-of-use. Headers, footers, margin and spacing control, page skipping, and numbering, as well as input file changing, are about all that is provided. For much work this is all that is needed.

All the function keys, Shifted, Alted, and Ctrl-ed, are used, as are many control keys. The functions assigned to the control keys, which are user configurable, are initially set to be similar to WordStar. Configuration is a strong point of PC-Write. The use of color (where available) for different word processing functions is well done. Pressing F1 produces a menu of all the function keys and other major operations. However, so many key functions are available that the help screen is bewilderingly complex. The designer seems to have leaned toward providing more function at the expense of simplicity and ease-of-use. I would hesitate to give this program to users unaccustomed to the complex keyboard of the IBM PC and expect them to be able to use PC-Write quickly. As an example, PC-Write has at least six or eight different cursor shapes—each denoting the present status of some function, such as insert (or pushright) mode, shift or control key status, or the current status of marking text. The variations are too many and too subtle for my taste.

Cursor movement is available in many ways—character, line, word, paragraph, or screen (up or down). However, the convention of which key to use is reversed from many other PC products. The meaning of the PageUp and PageDown keys can be reversed, but others cannot. All this leads to more confusion for inexperienced users.

The only bug I ran into was an inability to handle files that have linefeeds without always being accompanied by a carriage return. This interferes with the paragraph formatting and line delete functions and causes the program to loop, requiring a power-off reset to continue.

I found the product to be useful but too complex. In my rating scheme, ease-of-use is the most important attribute. The manual provided is well written, but it is hard to find information quickly. Topic headings in larger typefaces would be desirable.

PC Small-C 2.0

Company:
The Coriolis Company
Box 76, Clinton Corners,
NY 12514
Computer: IBM PC
Price: $35

Reviewed by Ronald G.Parsons

Over the years, *Dr. Dobb's Journal* has fostered and published the source code for a number of implementations of small versions of the C programming language. Ron Cain developed the original Small-C compiler for the 8080 microprocessor, published in *Dr. Dobb's Journal* in May 1980 and September 1980. Versions of this compiler were made available for CP/M (The Code Works) and IBM PC-DOS 1.1 (Caprock Systems). J. E. Hendrix developed a later version, Small-C 2.0, published in *Dr. Dobb's Journal* in December 1982 and January 1983. The product described in this review, PC Small-C 2.0, is a revision of Small-C 2.0 for IBM PC-DOS and Microsoft

MS-DOS (versions 2.0 and later).

PC Small-C 2.0 is not a complete implementation of the C language, nor does it claim to be. It does provide an inexpensive way to learn about and try the C language. It also provides, through the source code and libraries, an excellent way to learn about the structure and coding of high-level language compilers. The minimum requirements for using PC Small-C are an IBM PC (or equivalent) with 96K memory, one single-sided diskette drive, PC-DOS or MS-DOS 2.0 or later, and the 8088 PC Macro Assembler.

PC Small-C provides certain enhancements over Small-C 2.0, including redirection of input and output, additional library functions, hexadecimal and octal numeric constants, code generation optimized for the 8088 microprocessor, and automatic declaration of external function references. The authors claim that PC Small-C 2.0 programs are typically 30% smaller and twice as fast as equivalent Caprock Small-C:PC programs.

Features

PC Small-C 2.0 supports character and integer data types as global, local, external, and pointer variables, as well as one-dimensional arrays. *Not* supported are float, double, unsigned, long, short, or register variables, structures, or higher dimensional arrays. All the usual unary and binary operators are supported, as well as the selection operator. Statements supported include if, if ... else, while, do ... while, for, switch/case/default, break, continue, goto, and return. Compiler directives supported include #define, #include, and #ifdef ... #else ... #endif. In-line assembler code is supported by the #asm ... #endasm directive. A fairly complete set of I/O library routines is included, but open, close, read, and write system calls are not provided. This may cause some inconvenience in transporting C programs to this compiler. Many useful miscellaneous library routines include abs, isalpha, islower, isupper, peek, poke, tolower, and toupper. Several *IBM PC-specific* routines are also included, such as delay, sound, spkroff, and spkron. Command line argument support is available through argc, argv, and getarg.

	Small-C 2.0	Computer Innovations' Optimizing V2.10	Lattice C version 2.00
Compile/Link	66 sec	49 sec	46 sec
Execute	36 sec	14 sec	12 sec
.EXE size	14,369 bytes	16,015 bytes	19,618 bytes

Figure

Memory Utilization

PC Small-C 2.0 trades off memory availability (stack, data, and code are limited to a total of 64K) against speed of execution. For small C implementations, this is probably the proper tradeoff. Large programs requiring other data models should use a full implementation of C. In-line code sequences are favored for speed, and the use of segment override prefixes and the BP register is avoided. This is made possible by setting the data segment register, DS, and the stack segment register, SS, to the same value during initialization.

Compilation

The compilation process consists of a compiler phase, which produces assembler source code; an assembly phase, which produces an object module; and a link phase, which links the program object module with library object modules or separately compiled modules. Because of this assembly step and the slowness of the IBM assembler, the compilation process is somewhat lengthy. For instructional purposes, however, the easy availability of the generated machine code is invaluable. A compiler switch can cause the Small-C source code lines to be included as comments in the assembler source code.

Benchmark

No review of a C compiler seems to be complete without a benchmark using 10 iterations of the Sieve of Eratosthenes. The times in the figure (page 123) were obtained using an IBM PC XT running PC-DOS 2.1.

Conclusion

PC Small-C 2.0 is an excellent value to someone wanting to investigate or learn C or to learn how compilers are constructed. The small price includes a reference manual, the complete source and executable code, and even a remittance to Ron Cain and J. E. Hendrix.

DDJ

OF INTEREST

by Michael Wiesenberg

Turn jr into Papa

The **Drive Two Enhancement Package** from Rapport Corporation is an expansion kit for the IBM PCjr that enables it to run "virtually all" IBM PC software. For $675, you get a 360K floppy disk drive, parallel printer port, clock calendar with battery backup, and 128K memory (expandable to 512K). Optional accessories include an audio amplifier with speaker, to provide audio reproduction on any monitor, and a keyboard adapter cable.

Pascal on PC, jr, etc.

SBB Pascal, by Software Building Blocks, runs on the IBM PC and compatibles; with PC-DOS, MS-DOS, and CP/M-86; and also, they claim, on the PCjr. Execution time and code size compare favorably to other Pascal packages, according to benchmarks in their brochure. One of the best features is that you don't have to pay royalties on products using SBB code. The compiler package is $95. The development package is $350. The latter adds separate compilation, overlays, BCD arithmetic, 8087 support, sources for the runtime library, updates, and a copy of Watt's *Pocket Guide to Pascal* (Addison-Wesley). You get a debugger, editor, and editor source with both versions.

And Modula-2 for PC XT

Volition Systems, with versions of **Modula-2** already out for Apple, Sage, and IBM PC, now offers the language specifically for the XT—with utilities, access to any memory location, variables based at fixed memory addresses, RAM disk support, access to I/O ports, screen display mode, 8087 coprocessor support, exception handling, subprogram calls and overlays, random and sequential file access, directory operations, concurrency and interrupt handling, strings, and BCD arithmetic. For $395 you get both Modula-2 and Pascal compilers, a module library, the Advanced System Editor (ASE) for program development and text processing, a p-shell (Unix-like programming environment), utility programs, complete documentation, an overview for new users, a tutorial for those familiar with Pascal, installation guide, ASE User's Manual, programming hints, and Wirth's *Programming in Modula-2.*

C64 Disassembled

C-64 source, from Schnedler Systems, is the complete assembly language source for the Commodore 64's BASIC and Kernel ROMs, cross referenced and extensively commented. Even included are calls to routines not found in Commodore documentation. $29.95.

Cross-Assemble Almost Anything

2500 AD Software has cross assemblers that run on MS-DOS, PC-DOS, CP/M-80, CP/M-86, and Apple II with softcard. You can get all major 4- and 8-bit families, including F8/3870, COPS 400, NEC 7500, 87xx, 8051, 68xx, 65xx, 1802, NSC-800, Z-8, Z80, 8080, and 8085, and 16- and 32-bit microprocessors, including 8086/88, 80286, 16032, Z8000, and 6800. All packages include relocatable code, large file-handling capacity (buffer overflow to disk), macros, recursion, nesting, conditional assembly (to 248 levels of nesting), assembly time calculator, "include" files, listing control, hex file converter, cross reference table, and plain English error messages. $99.50 for 4- and 8-bit products, and $199.50 for 16- and 32-bit (plus $6.50 p. & h. each).

Full C for PC

c-systems has a full C compiler for the IBM PC and MS-DOS computers, supporting 8087 or assembly language subroutine floating point, and it addresses up to 1 Mb code or data. You get transcendental functions, ROM-able code generation, full featured liraries (including DOS 2.0 functions with automatic selection of DOS 1.1 and 2.0 calls), and program chaining. You also get **c-window**, a source-level debugger that displays source code while debugging, variable display and alteration, automatic command execution, and multiple breakpoints that can be set as function and line number. The package also includes a fast assembler. All of this will set you back but $199.

Keep it Clean

Mini-Vac keeps your computer running longer by removing dirt from between the keys and other inaccessible areas. You can also clean stereo, video, and other electronic equipment, cameras, and other areas most easily reached by a portable vacuum cleaner. This is not a compressed air unit that merely disperses pollutants; it sucks them into a replaceable cloth vacuum bag. You can also use Mini-Vac as a blower. It has two wands, two bristle brushes, and can be powered by DC or AC, with an optional adapter. It costs $29.95 and has a 90-day guarantee.

On-Line Desktop Accessories

Ever get a phone call in the middle of a computing session, need to jot down a note, and can't find paper and pencil? Or have to stop in the middle of editing to make a computation and can't find your calculator? **Sidekick** from Borland opens windows in your application program with electronic notepad and editor, calculator, appointment calendar, ASCII table, automatic dialing, metric conversion table, on-line help, and customizing capabilities to bring in special tools. You get into or out of Sidekick with a single keystroke. Currently available for IBM PC, XT, PCjr, and "true" compatibles, the product costs $49.95.

Moments to Remember

The Society for Computer Simulation holds its annual **Summer Computer Simulation Conference** July 23–27 at the Copley Plaza Hotel in Boston. Topics include continuous systems simulation languages, desktop simulation, supercomputers, artificial intelligence, and, in a special meeting of the Dutch Benelux Simulation Society, research topics on advanced information processing for simulation in the Benelux countries and demonstration of the Delft parallel processor. Conference registration is $155 for participants and members of sponsor or affiliate societies and $170 for nonmembers.

The American Association for Artificial Intelligence and the University of Texas at Austin Departments of Mathematics and Computer Sciences are jointly sponsoring a **conference on artificial intelligence research** August 6–10 at the Performing Arts Center at the University of Texas at Austin.

The Association for Computing Machinery (ACM) holds its **1984 annual conference** October 8–10 at the San Francisco Hilton. The main topic of discussion will be the fifth generation challenge from Japanese supercomputers.

The **Sixth Annual Forth Convention**, held by the Forth Interest Group (FIG), takes place November 16–17 at the Hyatt Palo Alto, with tutorials, exhibits, vendor booths, lectures, and discussions. FIG is a worldwide, nonprofit organization that has over 4700 members and 53 chapters.

Contact Points

ACM-84, The Fifth Generation Challenge, Box 32575, San Jose, CA 95152; (415) 948-6306.

American Association for Artificial Intelligence, 445 Burgess Drive, Menlo Park, CA 94025; (415) 328-3123.

Borland International, 4113 Scotts Valley Drive, Scotts Valley, CA 95066; (408) 438-8400.

c-systems, Box 3253, Fullerton, CA 92634; (714) 637-5362.

Forth Interest Group, Box 1105, San Carlos, CA 94070; FIG Hot Line (415) 962-8653.

Mini-Vac, Inc., Box 3981, Glendale, CA 91201; (213) 244-6777.

Rapport Corporation, 80 South Redwood Road, Suite 213, North Salt Lake, UT 84054; (801) 292-9454.

Society for Computer Simulation, Box 2228, La Jolla, CA 92038-2228; (619) 459-3888.

Software Building Blocks, Inc., Box 119, Ithaca, NY 14851-0119; (607) 272-2807.

Schnedler Systems, 1501 N. Ivanhoe, Dept. NR, Arlington, VA 22205; (703) 237-4796.

2500 AD Software, Inc., 17200 East Ohio Drive, Aurora, CO 80017; (303) 752-4382.

Volition Systems, Box 1236, Del Mar, CA 92014; (619) 481-2286.

DDJ

Special Forth Issue

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

#93 September 1984

$2.95 (3.50 Canada)

Forth Cross Compiler for the MC68000

File Maintainance in Forth

Forth and the Fast Fourier Transform

Computing With Streams

09

0 38351 16562

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*

Production
Design Director *Fred Fehlau*
Art Director *Shelley Rae Doeden*
Production Manager *Detta Penna*
Production Assistant *Alida Hinton*
Cover *Isaac Victor Kerlow*

Advertising
Advertising Sales *Alice Hinton, Walter Andrzejewski*

Circulation
Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.
Publisher and Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

September 1984
Volume 9, Issue 9

CONTENTS

This Month & Next

With the growing acceptance of Forth-83, we expect to see more energy being put into designing useful Forth applications and less into discussion of how Forth should be implemented. This year's Forth issue reflects that expectation. A quick perusal of the facing page will reveal a Fast Fourier Transform package, a cross compiler for the MC68000, and a file maintenance package. There is also an article which uses Forth words to examine some programming issues. That the 83 Standard did not cover everything is also in evidence. You might want to examine the FVG floating-point extension to see what some folks are doing until the next meeting of the Forth Standards Team.

One theme that emerged this month, in both the FFT article and in the C/Unix Programmer's Notebook, was the use of complex numbers in programs. Those interested in more detail on the topic should stay tuned for an article next month that discusses complex numbers in greater detail. Also next month, we will feature a Unix-like generalized regular expression parser. In addition to all our regular columns, we will introduce The Software Designer, a new column focusing on a wide range of programming issues.

A Quick Fix for Subscription Bugs

By September 7, 1984, all subscribers, current and former, should have received a letter from our Circulation Department explaining that we recently contracted an excellent West Coast fulfillment service in an effort to serve you even better. We are anxious to clear up any problems you may have been having with your subscription, and we are giving you a golden opportunity to have those problems fixed. Accompanying the letter is a form for documenting any difficulties you are experiencing. If you need your address changed, your expiration date corrected, missing issues supplied, or some other adjustment, this is the best way to ensure quick action. In fact, if you have *not* received the letter yet (that may tell you something), you should call our fulfillment firm toll-free at 1-800-321-3333 (inside CA dial 1-619-485-6535); or drop Circulation a note (preferably with a magazine label) at 2464 Embarcadero Way, Palo Alto, CA 94303.

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to remarks to the editors concerning accuracy and relevance of manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness. The referees who contributed to this month's issue are:

Ray Duncan, *DDJ* Contributing Editor
Kim Harris, Dysan Corporation
William Ragsdale, President, Dorado Systems

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

File Maintenance in Forth 24
by Ray Cadmus
This set of Forth words lets one build a simple description of a data file and its associated screen format, and then add and change the file's records.

Forth and the Fast Fourier Transform 34
by Joe Barnhart
Adapting a floating-point algorithm to Forth's fixed-point math, this Forth-83 FFT package can be used for a variety of applications.

Computing with Streams 50
by L. L. Odette
The author develops a set of Forth words to experiment with concepts of data-flow oriented computation, using a data structure called a stream.

A Forth Native-Code Cross Compiler for the MC68000 68
by Ramon Buvel
With this cross compiler, an experimenter can use an 8-bit computer to generate machine code for the 68000 from Forth source code.

The FVG Standard Floating-Point Extension 110
by Ray Duncan and Martin Tracy
The Forth Vendor's Group has recently adopted this Forth floating-point extension, which might well find its way into future Forth packages.

DEPARTMENTS

Editorial 6

Letters 8

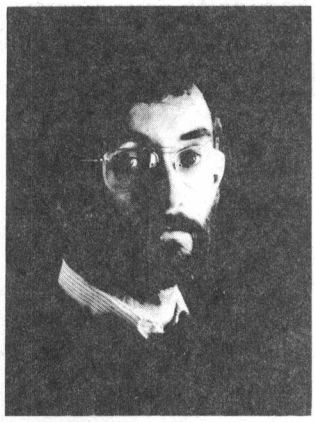
Dr. Dobb's Clinic 12
Lose That Ugly FAT; A RAM-Drive is Not a Sheep Roundup; I'm OK, UART Confused

CP/M Exchange 18
CP/M Plus: Interbank Memory Moves Without DMA; Z80ASM: An Assembly Language Development System

C/Unix Programmer's Notebook 116
Ways to make C more powerful and flexible, including such things as complex types, pointers to operators, and variable-length automatic arrays

Of Interest 126

EDITORIAL



In *A Programmer's Notebook: Utilities for CP/M-80*, our resident intern Dave Cortesi argues that programming is a craft. "Like [woodworking, pottery or blacksmithing], programming has the goal of producing utilitarian objects. Like those crafts, its practice calls for the application of skills developed by long practice, and for the application of the artisan's own good taste and imagination to the elaboration of conventional designs. And like any good craftperson, the good programmer strives to reach beyond mere utility to efficiency, elegance, and beauty."

Not everyone agrees; C.A.R. Hoare (in *IEEE Software*, April 1984) wants to claim that programming "has transcended its origins as a craft, has avoided the temptation to form itself into a priesthood, and can now be regarded as a fully-fledged engineering profession." I think Dave is closer to the truth, and that the forces pushing programming in the direction of an engineering discipline are the same forces that would maintain the priesthood. Programming as a craft is at least a good metaphor, and provides a context for discussing programming's design aspect.

The programmer's taste and imagination produce different approaches to software design, such as whether the programmer works alone or on a team, writes in C or in Ada, feels like a craftsperson or an engineer. "One does not use structural engineering analysis to build a sandcastle," Hoare says, "But neither does one choose the prize-winning builder of sandcastles as an architect for a tower block of offices in a city." Sandcastles vs skyscrapers? Hoare's distinction shows a bias toward the engineering style; I think that a more meaningful distinction is that between those problems that an individual can't solve (and that therefore require an engineering-team approach) and those that only an individual can solve. Maximizing efficiency, for example. Efficiency rewards tricky programming, and tricks are anathema to the engineering team approach.

The craft metaphor fits in other ways, too. As is typical of artisans, programmers often don't benefit from their own craft. Software designers routinely work with inadequate, ill-designed tools. Hoare would applaud "when a software engineer designs a [software tool] that can be fully defined in 20 pages whose rival product has been inadequately defined in 100 pages . . . ."

Perhaps the designers of software tools should stop designing "utilities" and start designing "efficiencies," "elegances," and "beauties."

Whether we call programming a craft, an art, an applied science, an engineering discipline, black magic, or a religion (you thought this editorial had nothing to do with Forth, didn't you?), we should not lose sight of the importance of its design element. Starting next month in our new column The Software Designer, we will call upon some working software designers to help us examine what makes good design, and why that question has different answers in different task domains. We'll look into different programming styles and ask why some programmers work in Forth or C while others use Ada or Modula-2. We'll ask designers what (currently nonexistent) tools they would most like to have. We'll examine software design in diverse environments: under Unix, on a Mac (no, not "for a Mac, on a Lisa"), and in logic (i.e., Prolog). And we'll examine the kind of programming this magazine was founded to foster back in 1976: highly-efficient code, or as we put it then, "running light without overbyte."

Michael Swaine

Michael Swaine

Making Languages English Independent

Dear *DDJ*,
I was delighted to see the article on Chinese Forth in the June 1984 issue of *DDJ*. In many ways this is one of the most significant articles you have ever published because it recognizes for the first time that not everybody, or even a majority of the world's population, speaks English, and that to use a programming language based on English is to deny the non-English speaking majority easy access to the pleasures of personal computing.

There are two main approaches to rectifying the problem. One, as exemplified by APL, is to eschew natural language words altogether in favor of ideographic symbols. The second approach, used by Timothy Huang, is to take an existing language that allows redefinition of function names, and then rename them in the new language.

Probably the easiest computer language in which to do this is SAM76, described in your pages a few years back. It actually possesses a primitive function '@cn' that can be used to change the name of any primitive function. Using it, I have created a version of SAM76 that has all its primitives named in Polish. A new language called HERPES that I am currently working on also includes this feature.

It is a pity more was not said about the problems of printing Chinese text. For some natural languages, the problem of printing and displaying texts may easily exceed that of changing the keywords in the programming language. Chinese presents particular problems both in the complexity of the characters and the sheer quantity of them. Others with a lesser order of complexity are Arabic (with context-dependent letter shapes) and the Devanagari script (used for Sanskrit, Hindi and other tongues of North India) where two or three letters are regularly combined to form composite characters, even across the word boundary. I am currently struggling with the problem of printing Sanskrit on a Gemini 10X and it is not a trivial exercise.

Earlier I mentioned SAM76. Does anybody else use it? I haven't heard anything about it for about five years now and wondered if I were the only user. I find it unexcelled for programs manipulating character and text data, in fact most things except number crunching, where it is slow and suffers from awkward syntax, though the unlimited precision is often useful. Finally, for the linguists among you, I enclose a piece of text printed on a Gemini 10X using a polyglot word processing program written in SAM76 (see Figure 1, below). What language is it? (*No* prizes!)

Sincerely,
Greg Trice
1131 Sandhurst Circle #111
Scarborough Ontario
M1V 1V5, Canada

Dear *DDJ*:
I was very impressed with the ideas presented by Timothy Huang in his article "First Chinese Forth—a Double-Headed Approach" which appeared in *DDJ* No. 92 (June, 1984). Mr Huang has brought attention to a topic that merits further discussion and exploration. In my view, the problem of natural language barriers to the use of computer languages should be attacked at several levels. One solution is the use of translated keywords, as supported by Chinese Forth. For introductory teaching applications, instructions given to the computer should be as natural as possible. Thus, BASIC, Logo and other teaching languages should probably have translated command-sequences whereever English is sufficiently foreign to inhibit the use of the computer. However, software portability would become infeasible if each community utilized its own keywords to the exclusion of all others. This dilemma requires its own solution.

If computer language keywords were translated for each of the major natural languages, we would have a virtual tower of Babel in the computer world. At the moment, we have many different programming languages, but these languages are typically driven only with English-derived instructions/keywords. Programs created under Chinese Forth would not only be foreign to English speakers, but would also be foreign to Indonesian Forth, French Forth or even Japanese Forth programmers. Under such circumstances, programming communities would factionalize along natural language lines. Sharing software would be less possible, and the literature of any specific community would be much less valuable to other communities. This would magnify the problem of repetition of effort and would most seriously hurt nations that are underde-

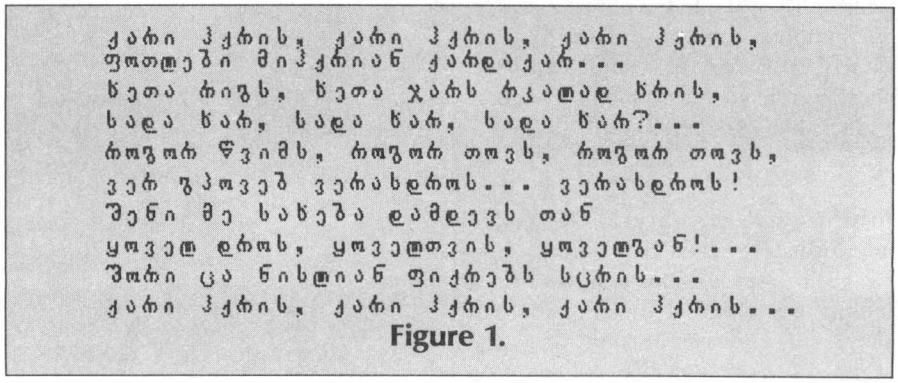


Figure 1.

veloped in the computer sense.

One solution to the problem might be to restrict programming to languages that lack any English keywords. This remarkably strong restriction essentially leaves only APL or direct, machine-code programming. APL is suitable for a large number of applications, but isn't for everyone. Furthermore, asking a whole world of programmers to work with one language is not reasonable in the least.

On the other hand, an important lesson can be gained from looking at APL's instruction mnemonics. These symbols suggest a compromise between keyword translation and the use of APL. We could create a set of symbolic keywords for a variety of languages; these keywords would bear no English dependence. Such keywords would ensure some degree of machine readable portability between communities, and simultaneously allow listings in the literature to become more accessible.

In order to demonstrate this idea, I devised a set of symbols for the C programming language (see Table 1, page 10). These symbols replace the compiler keywords and preprocessor commands as listed in *The C Programming Language*. Since C has no intrinsic functions, there was no need to create symbols for a large set of such internal functions. (Creating symbols for the standard input-output library names and definitions is, I believe, beyond the scope of this discussion).

The symbols presented in Table 1 themselves represent a compromise. They were selected from among those available on an IBM Personal Computer (or compatible). This was done so that this software concept could be used immediately without the need to invent new character generators, as with APL. However, these symbols are reasonably good and could be a practical symbolic replacement for C's English keywords. Rationale for the choice of the various symbols is included in Table 2 (page 10).

Many programmers will find the symbolic keyword alternative inferior to the use of natural language keywords. However, the symbolic language concept is still useful. It offers a means by which programmers can share software. Once they have the symbolic form of the program, a straightforward translator could be used to convert the code to their own particular keyword syntax. A reverse translator could be used before software is distributed (or published). Yet another translator could preprocess foreign keywords (into English equivalents) so that standard compilers could continue to be used.

The symbolic language does not help with documentation aspects of portability. Software documentation would have to be translated, as would command prompts and other text within the code itself. (I see no clear way to solve this problem, except by encouraging documentation in multiple natural languages.)

In this brief commentary, I've addressed another approach designed both to open the computer world to non-English speakers and to have an impact on keyword translation to software portability. I hope that others will feel free to comment in order to continue the dialog in this area.

Sincerely,

Anthony Skjellum
Pyramid Systems, Inc.
1695 Shenandoah Road
San Marino, CA 91108

DDJ

Symbolic Replacements for C Language elements

C Keywords

±	(int)	Γ	(char)	Φ	(float)	Θ	(double)	ϵ	(struct)
∩	(union)	‖	(long)	¢	(short)	╪	(unsigned)	α	(auto)
▵	(extern)	φ	(register)	ƒ	(typedef)	¥	(static)	↑	(goto)
↓	(return)	╬	(sizeof)	↨	(break)	↕	(continue)	¿	(if)
♦	(else)	Σ	(for)	δ	(do)	Ω	(while)	≡	(switch)
→	(case)	←	(default)	♫	(entry)				

C Preprocessor controls

#¿	(#if)	#¿≡	(#ifdef)
#¿!≡	(#ifndef)	#♦	(#else)
#?	(#endif)	#£	(#line)
#≡	(#define)	#!≡	(#undef)
#¶	(#include)		

Miscellaneous

§( )	main( )

Table 1.

Symbol/IBM Chr. #	Keyword	Rationale
± 241	(int)	Symbol indicates a signed data type
Γ 226	(char)	Gamma is the third Greek letter, analogous to 'c' of char
Φ 232	(float)	No specific reason
Θ 233	(double)	No specific reason
ϵ 238	(struct)	Epsilon looks like mathematical 'belongs'
∩ 239	(union)	Intersection sign is opposite of union sign
‖ 186	(long)	Double bar through a negative sign
¢ 155	(short)	No specific reason
╪ 215	(unsigned)	Double bar through a negative sign
α 224	(auto)	Alpha is the first Greek letter, analogous to 'a' of auto
▵ 127	(extern)	Character points upward; external data normally at top
φ 237	(register)	No specific reason
ƒ 159	(typedef)	Like an f sometimes used in mathematical definition
¥ 157	(static)	Symbol is similar to the symbol for electrical ground
↑ 24	(goto)	Upward arrow to symbolize a jump or goto
↓ 25	(return)	Downward arrow symbolizes going to end of block ie return
╬ 206	(sizeof)	Symbol looks like measurement device
↨ 23	(break)	Two way arrow with bottom supposed to indicate break
↕ 18	(continue)	Two way arrow less bottom indicates continuation
¿ 168	(if)	Upside down question-mark indicates interrogative
♦ 4	(else)	Diamond symbolizes decision-diamond from flowcharting
Σ 228	(for)	Mathematical summation symbol; often done with for loops
δ 235	(do)	Delta is fourth Greek letter, analogous to 'd' of do
Ω 234	(while)	No specific reason
≡ 15	(switch)	Symbol looks like a mechanical switch
→ 26	(case)	Right pointing arrow indicates item to follow
← 27	(default)	Left pointing arrow opposite of case ie default
♫ 14	(entry)	No specific reason

C Preprocessor controls

#if	#¿	Symbolizes interrogative
#ifdef	#¿≡	Reads 'if defined'
#ifndef	#¿!≡	Reads 'if not defined'
#else	#♦	Decision diamond symbolizes else
#endif	#?	Endif is opposite symbol to if
#line	#£	Pound sign supposed to look like a fancy 'l' for line
#define	#≡	Mathematical define symbol
#undef	#!≡	Reads 'un-define'
#include	#¶	Paragraph mark since this causes more text to be read

Miscellaneous

§( )	main( )	Section mark to indicate main program section

Table 2.

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

Lose That Ugly FAT

"Just as your mailbag ran dry," writes Chet Floyd, "the FAT on my IBM PC XT's hard disk was clobbered during a Pascal compile." Uh, sorry, Chet. And the rest of you take warning—keep those letters coming or make lots of backups! But to return to Chet's story . . .

"It was a disastrous encounter with what I suspect is a bug in PC-DOS 2.00. PAS1 used the first sector of the FAT as a directory. I do not have a definite answer to the problem, just a couple of theories you may wish to explore or explode.

"First, the players: IBM-DOS 2.0, MS Pascal 3.13, a nearly full hard disk, and 640K of memory busy with an AST memory disk and spooler, Btrieve, and MultiJob running 250K and 128K partitions. The compiler was in the larger partition, Personal Editor in the smaller. I've used this configuration for at least seven months, with 3.13 replacing the 1.0 compiler three months ago. The conglomeration has worked flawlessly.

"Then the crash occurred, with PAS1 vaguely indicating a 'File Access Error.' When I got around to inspecting the FAT, it was obvious that the first sector had been used for the directory entries of the work files PAS1 builds; there were three entries there, and the rest of the sector was set to zeros. Of course, the backup copy of the FAT was similarly bent.

"This caused the 300-odd clusters controlled by this FAT sector to 'vanish.' About 1.5 megabytes instantly became free; this didn't include the files controlled by directories that had resided in those 300 clusters.

"The theory I favor is that DOS cannot recover the multilevel error that results when both the current directory and the device are full. When PAS1 tried to create the first work file, DOS inspected the FAT and found no space, leaving the disk sitting at the first sector. But this error was perhaps not passed back, and DOS just used the current disk location to create another cluster for the directory. An alternative theory is that MS Pascal does its own disk I/O.

"Upon notifying Microsoft, I was told of a new version of MS Pascal, 3.2, which is supposed to have better disk-full recovery. I ordered it immediately, because *new version* is the code word for *your bug is fixed.* I asked why I, as a registered ISV and MS Pascal user, hadn't been notified that version 3.13 had been superseded. 'We don't have the database built yet, but we're trying to get to it.' So much for ISV support.

"I also contacted B&L Computing, the developer of MultiJob. To his credit, Bob Hudson, who wrote MultiJob, said he wished he could absolutely rule his code out of responsibility for the crash, but wouldn't, because of the black-box nature of parts of DOS. However, I am inclined to rule out MultiJob as the culprit.

"Although my backup procedures ensured that I would suffer no permanent loss of data, I was surprised at how much I had depended on my ability to reconstruct files as an excuse not to back them up. Reconstruction took several days and was traumatic enough to dissuade me from trying to force the condition or test my theories in any other way. I consider myself lucky to have been able to recover, because my backup strategy had never been subjected to the acid test of actual recovery."

Comments, anyone?

A RAM-Drive is not a Sheep Roundup

Reader Leonard Schwab was interested in our report on installing an Electra-Logics Quasi-Disk. He had just completed a more challenging installation. "Recently, I obtained a CompuPro M-Drive-H board for my six-year-old IMSAI VDP-80.

"CompuPro provides very little information about installing the M-Drive in a non-Godbout system. The documentation talks about installation software that did not come with my board. There is a model driver routine in the manual, but some crucial points are omitted. You must analyze the model driver to discover the nature of the data required by the board during accesses, and nothing is said about the technicalities of formatting the board on a cold start. Needless to say, this only sweetened the challenge."

Schwab eventually ended up rewriting all the disk-handling code of his Fischer-Freitas BIOS. In the process, he was able to add automatic density-sensing for his Persci floppies. He found that he could copy the system image to the M-drive during a cold start, so that warm starts became almost instantaneous, and he adjusted the system so that submit jobs would run with no disk accesses at all. These actions transformed his system, he says.

If anyone else is having trouble integrating the CompuPro M-drive into a non-CompuPro system, Schwab is willing to advise. Send us a note and we'll pass it on.

I'm OK, UART Confused

Yeah (sigh), wouldn't it be nice if CompuPro's manuals were as good as the boards they describe? We just spent 15 hours integrating an Interfacer 4 into our system. It replaces an antique Interfacer 1, one of whose two ports died when we misjumpered it.

Fifteen hours is too damn much time. Now, it isn't all CompuPro's fault. We spent the largest part of the time trying to comprehend the interfaces between the software and the board, between the board and the RS-232 connection, and between RS-232 and the devices that we wanted to con-

nect to the system.

But the manual could have been a lot more help. For instance, it could have used a consistent way of referring to the three serial ports on the board. If you have an Interfacer 4, memorize this list:

- User 1 or User 5 is Port C; it's on the right.
- User 2 or User 6 is Port B; it's in the middle.
- User 3 or User 7 is Port A; it's on the left.

Simple, isn't it? (No!) It would have been even nicer if the writer had used standard terminology for the RS-232 configurations: DCE and DTE, not "master" and "slave" (now, which was "master"? Oh yeah, the terminal—I mean, DTE . . .).

Because the serial ports are the same on at least three CompuPro boards (the System Support 1 and the Interfacers 3 and 4), we documented what we learned. We present it here for the benefit of any reader that needs it.

DCE and DTE

There are two actors in a standard RS-232 connection. They are called the data terminal equipment, or DTE, and the data communications equipment, or DCE. The names are standard-ese for "terminal" and "modem"; that is, a terminal-type device normally plays the DTE role, and a modem-type device plays the DCE role.

The RS-232 interface is used between many sorts of devices. Sometimes it isn't obvious what role a device should be expected to play. In general, a device that is an original source of (or a final sink for) data, will play the DTE role. A printer, a plotter, a digitizing tablet, or, of course, a terminal, will play the part of the DTE. The other actor in the connection—usually a computer or a modem—must take the role of DCE.

A personal computer may be called on to play either role, depending on the application. When the computer is connected to a device that is clearly a DTE, the computer must act as the DCE, but when the computer is hooked to a modem, it must act as the DTE. When two computers are connected directly to each other, it doesn't matter which one plays which role—but both cannot play the same role.

One more point: the DCE actor normally is equipped with a female DB-25 socket, while the DTE actor is normally equipped with a male DB-25 plug. The presence of a female connector should indicate that a device is wired to play the DCE role (check your modem), and the presence of a male plug should indicate that the device is meant for use as a DTE.

This rule is followed most of the time with modems and terminals, but you can't depend on it when you are looking at the back of a computer.

Signal Lines

In most applications, there are only seven significant lines in the RS-232 interface (aside from ground and signal-ground lines). Here they are with their standard pin numbers in the DB-25 connector:

- Pin 2 carries data from DTE to DCE.

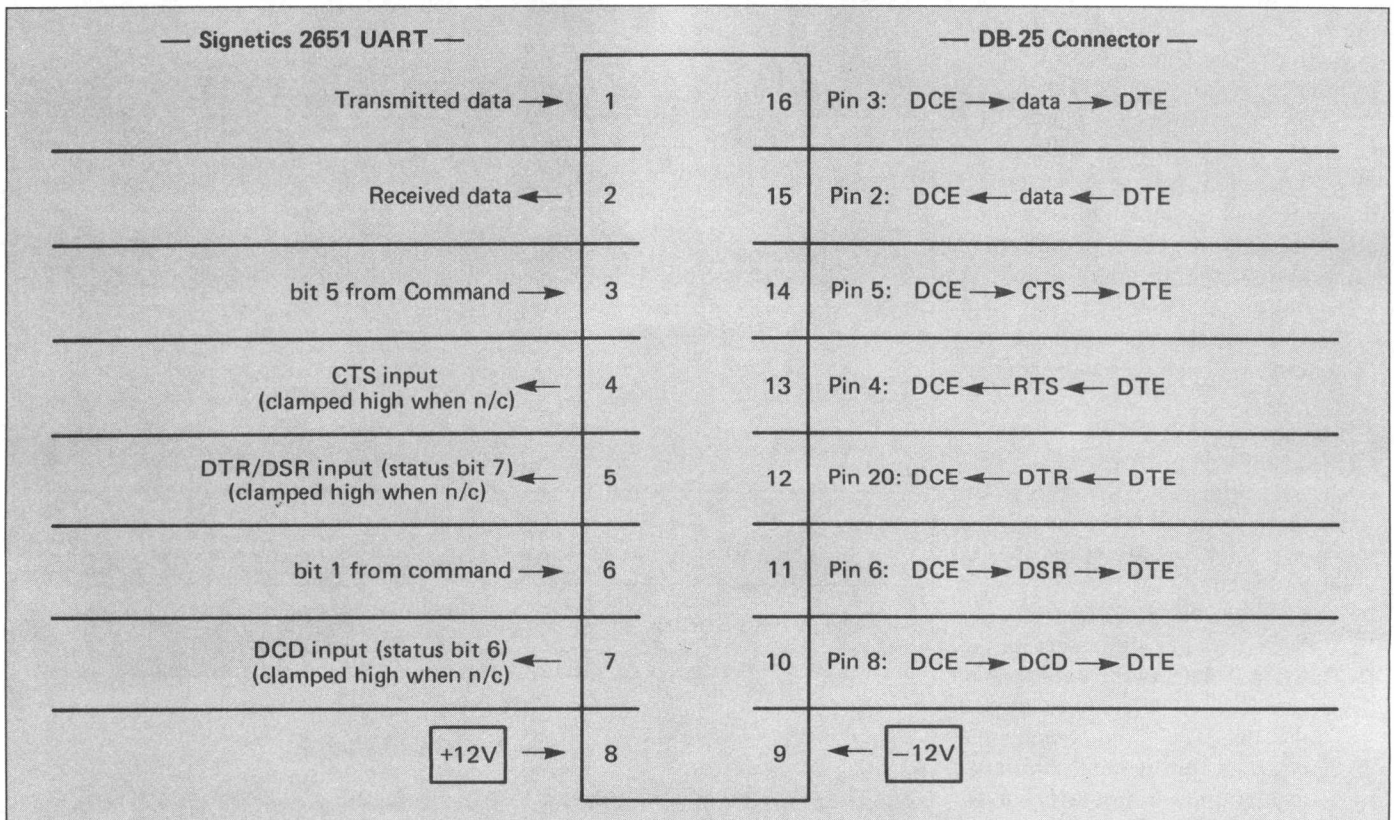


Figure 1 The DIP-socket interface between UART and DB-25.

- Pin 3 carries data from DCE to DTE.
- Pin 4, Request to Send (RTS), is asserted by the DTE when it has data to transmit.
- Pin 5, Clear to Send (CTS), is asserted by the DCE when it is OK for the *DTE* to send.
- Pin 6, Data Set Ready (DSR), is asserted by the DCE when it is on-line and ready.
- Pin 8, Data Carrier Detect (DCD), is asserted by the DCE (a modem) when a connection is established over the telephone line.
- Pin 20, Data Terminal Ready (DTR), is asserted by the DTE when it is ready and on-line.

Let's repeat that list from the viewpoint of each device. As the DTE (the terminal, printer, and so on) sees the interface,

- It transmits on pin 2.
- It listens on pin 3.
- It asserts pin 4, RTS, when it wants to send.
- It awaits pin 5, CTS, before sending.
- It asserts pin 20, DTR, as long as it is ready for operation.

In addition, a true terminal may use pin 8, DCD, as a qualifier of its other operations. The Diablo 1650 KSR terminal, for example, lights a green "proceed" light when it detects DCD true. Many DTE devices will use pin 20, DTR, as a sort of reverse-CTS, dropping it when some temporary condition puts a halt to their operations. A printer, for example, may drop DTR when it is out of paper.

Now, from the standpoint of the DCE actor, we have these viewpoints:

- It listens on pin 2.
- It transmits on pin 3.
- It asserts pin 5, CTS, when it can take data from the DTE.
- It asserts pin 6, DSR, when it is on line and ready.

A true modem will assert pin 8, DCD, when it detects the data tone on the phone line. A half-duplex modem *will take the DTE's* signal on pin 4, RTS, as a sign that it should initiate a line turnaround in preparation for transmission and will not assert pin 5, CTS, until the turnaround is complete.

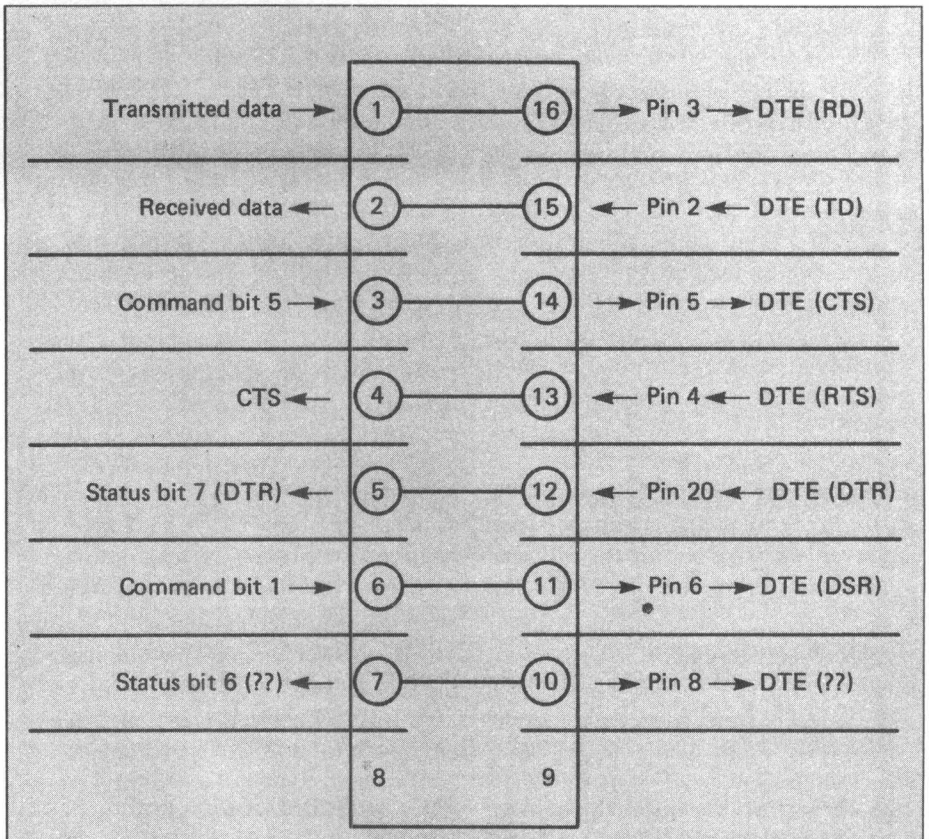


Figure 2 The effect of a DIP shunt is to make the UART play the DCE role.

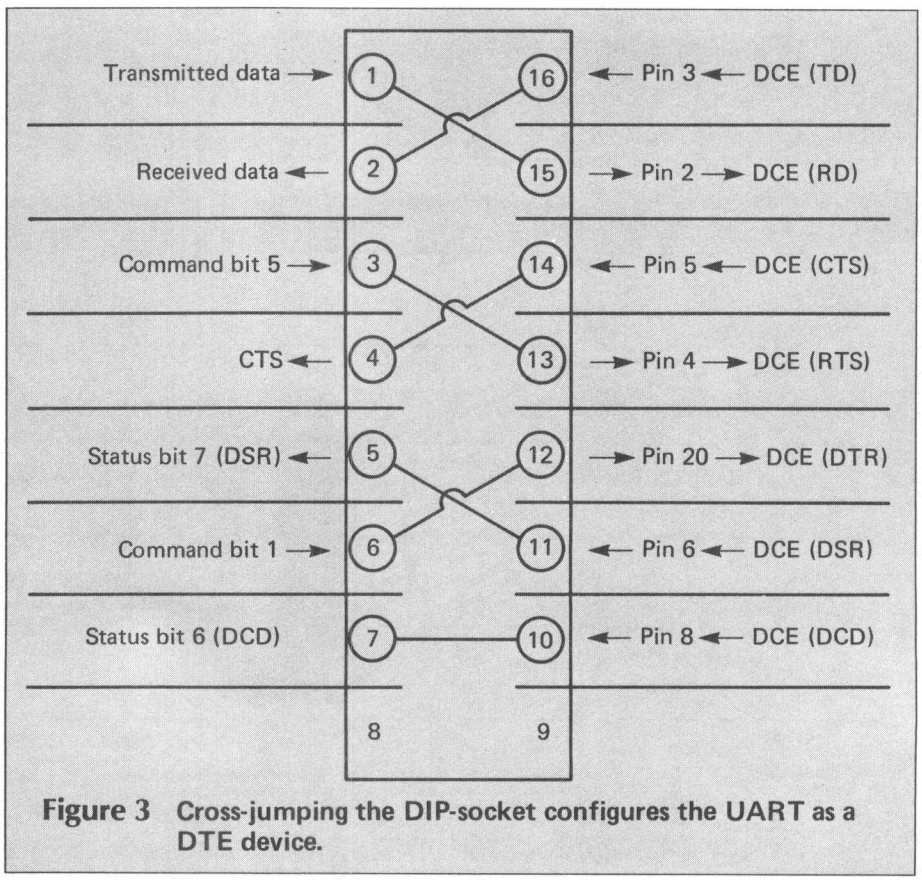


Figure 3 Cross-jumping the DIP-socket configures the UART as a DTE device.

A personal computer playing the DCE role has no phone line to listen to and will often use these lines in a simpler way. It might, for example, simply short the DTE's μ.
pin 5, CTS, supplying an instant "Clear to Send" without effort.

The UART Connection

The 2651 UART is used on the CompuPro System Support 1, Interfacer 3, and Interfacer 4 boards. In each case, its connection to the RS-232 link is the same. There is a single, 16-pin, DIP socket for each UART. Seven lines are brought from the UART to pins 1 – 7 of the socket, while seven lines are brought from the DB-25 connector to pins 16 – 10, the facing positions on the socket. The arrangement is shown by the diagram in Figure 1 (page 14).

Let's run down the UART's signals as they appear on socket pins 1 – 7:

- Socket pin 1 carries data transmitted from the computer
- Socket pin 2 receives data from the connected device
- Socket pin 3 carries bit 5 from the command last written to the UART, translated to RS-232 voltage levels
- Socket pin 4 is presented to the UART as Clear to Send signal
- Socket pin 5 is passed through to become bit 7 of the UART status byte
- Socket pin 6 carries bit 1 from the command last written to the UART, at RS-232 levels
- Socket pin 7 is passed through to bit 6 of the UART status byte

Two of the socket pins (3 and 6) are driven directly by command bits that your software writes to the UART. These bits pass through an even number of level inversions on their way to the socket; hence, the socket levels reflect the bit values in the command. A one bit in the command will appear as a positive voltage at the socket; a zero bit will apppear as a negative voltage.

Two of the pins (5 and 7) are passed through to the UART status byte (and have no other apparent effect on UART operation). Like the command bits, these status bits reflect the voltages applied—a positive voltage will produce a one bit, a negative voltage appears as a zero bit.

The Clear to Send input on socket pin 4, the status input on pin 7, and (on the Interfacer 4 but not on the System Suppport 1) the status input on pin 5 are all pulled up to positive voltage levels by a resistor. Thus, if these pins are not driven, they will appear positive.

Configuring the Port

You use this DIP socket to connect the UART to the DB-25 connector. In doing so, you configure that port as either a DCE or a DTE. This configuration is determined by the way that pins 1 – 7 are wired across to 16 – 10. Pins 8 and 9 on the socket are supplied with constant voltages for your convenience; you can use them as a means of tying a DB-25 pin to a fixed level. In a simple connection, you can use the socket as a means of connecting RTS back to CTS or for doing other tricky things. Here, however, we will examine the full configurations for the DCE and DTE roles.

The DCE Configuration

Examine Figure 2 (page 15). It shows the effect of installing a DIP shunt in

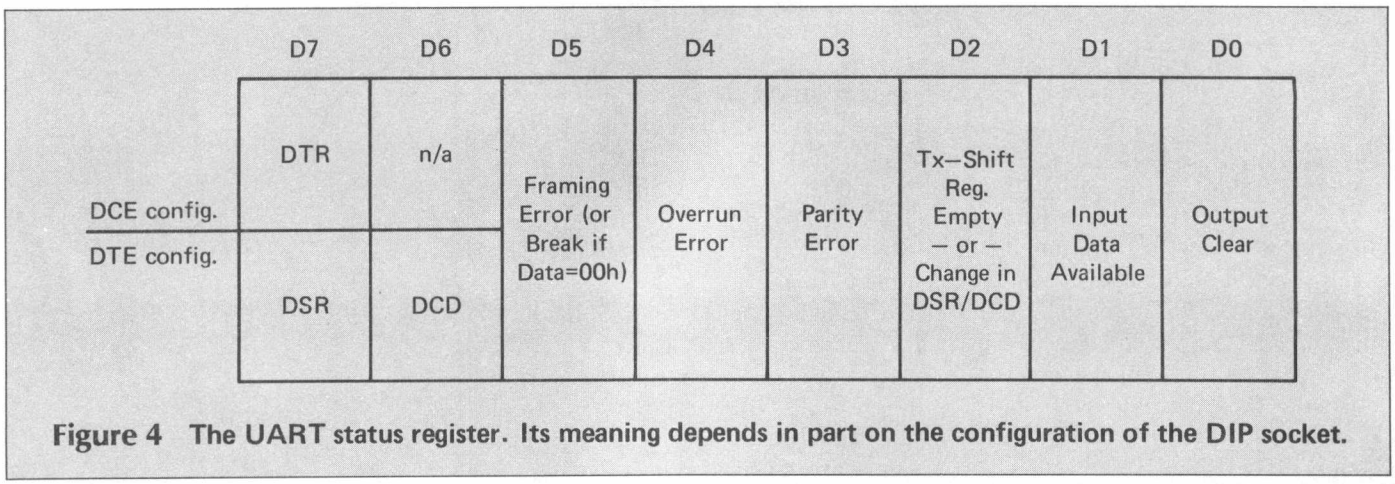


Figure 4 The UART status register. Its meaning depends in part on the configuration of the DIP socket.

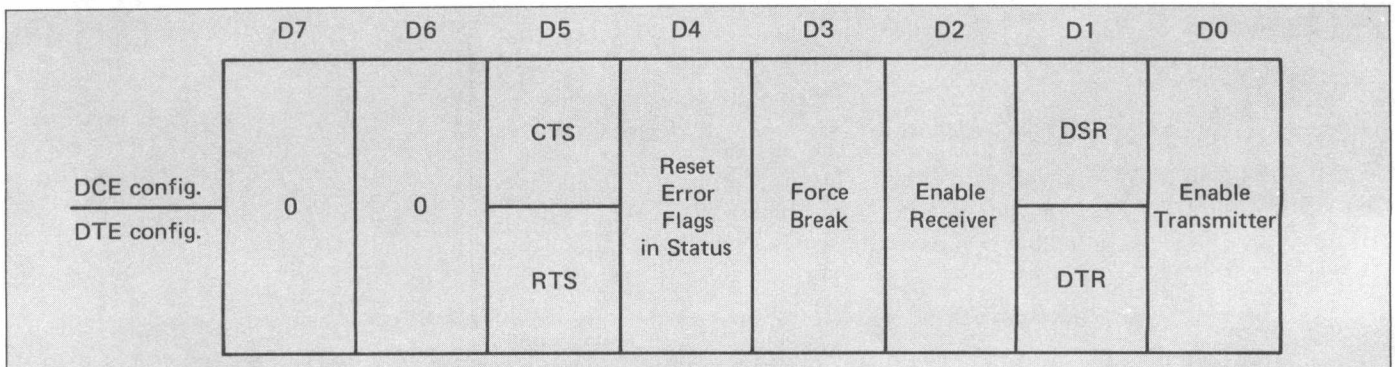


Figure 5 The UART command register. Its meanings depend in part on the configuration of the DIP socket.

the socket. Notice that the conductor between pins 8 and 9 *must be broken*; if you use a new, unmodified DIP shunt, you will be placing a direct short from +12 volts to −12 volts. Be careful about this; your CompuPro board may *have been* delivered with unmodified DIP shunts in these sockets.

The result of shunting each pin 1 – 7 to its facing pin 16 – 10, as shown in Figure 2, is to make this port a DCE. That is, the UART will be transmitting on pin 3, receiving on pin 2. You have direct control of RS-232 pins 5, CTS, and 6, DSR, through the UART command byte. Pin 8, DCD, will be held to a positive level by the pull-up resistor. The DTE's signal on pin 20, RTS, will be available in the UART status byte for your device driver to test.

There are a couple of anomalies in the configuration of Figure 2. The DTE's signal on pin 4, RTS, will be passed to the UART as "clear to send." It seems to cause no problem. And bit 6 of the status byte will always be one, since socket pin 7 is driven by neither device and hence will be pulled high. Nevertheless, this configuration works quite well as a DCE actor.

The DTE Configuration

Now turn to Figure 3 (page 15). This is how the DIP socket must be wired if the port is to be configured as a DTE. You must solder up this wiring using a DIP header. Furthermore, when configuring a port as a DTE you should bring it out on the back of the computer with a male plug. If you don't, you'll need an RS-232 "sexchanger" cable to connect it to the female socket on a standard DCE device. Alas, all the CompuPro-supplied interface cables terminate in female sockets at all positions, so you'll have to make up your own ribbon cable.

Be that as it may, let's consider the effect of the wiring shown in Figure 3 (page 15). The UART transmits on DB-25 pin 2 and receives on pin 3, as a DTE device ought to do. Your software can control the DTE signals of pin 4, RTS, and pin 20, DTR, through bits of the UART command byte. The attached DCE device's signals on pin 6 (DSR) and pin 8 (DCD) will be available in the UART status byte. The DCE's pin 5 signal, CTS, will be delivered to the UART as "Clear to Send."

UART Command and Status

Figure 4 (page 16) summarizes the status byte presented by the 2651 UART under these configurations. Six of the bits reflect the UART's operations. Bit D7 reflects the voltage state of socket pin 5; that is, either pin 20, DTR, or pin 6, DSR, depending on the socket configuration. Either way, it means "other device ready."

Status bit D6 reflects a modem's pin 8 signal, DCD, when the socket is configured as a DTE. If the socket is configured as a DCE, this bit will always be high. Of course, you could modify the wiring of Figure 2 (page 15) to bring another signal to this bit.

Figure 5 (page 17) summarizes the use of the UART command byte. A command can be written to the UART at any time after it has been initialized; it isn't necessary to write the two mode bytes before every command. Four of the command bits control UART operations. Two of them are simply passed through to the configuration socket and from there to the DB-25 connector, however.

Command bit D5 controls the level at socket pin 3. When the port is configured as a DCE, this bit will be seen by the DTE as a CTS signal. In most applications it should be set to 1, because most DTE devices want a positive CTS (but some may not). When set up as a DTE, command bit D5 controls RS-232 pin 4, RTS. Most DCEs expect this signal to be positive and will turn it around as pin 5, CTS.

Command bit D1 controls the level at socket pin 6 and, hence, when the port is a DCE, it controls the level of pin 6, DSR. Most DTE devices expect this signal to be high when their modems are on-line. When the port is DTE, command bit D1 controls the RS-232 signal pin 20, DTR. Most modems expect this line to be positive when their terminals are on-line.

Thus, in most applications, both command bits should be written as 1 bits. This may not be true in all cases, however, because the world of RS-232 devices contains more exceptions than it does normal cases. At least, once you understand the requirements of the attached device, you can satisfy them through some appropriate wiring of the configuration socket and some setting of the command bits. **DDJ**

by Robert Blum

Of all the modules needed to form a complete banked version of CP/M Plus, the MOVE module is the smallest in size but perhaps the most difficult to debug. The three subroutines residing in the MOVE module form the nucleus of logic to use banked memory. The close relationship that exists between these routines and the hardware can be the cause of many mysterious system crashes, especially when you bring up CP/M Plus for the first time.

CP/M Plus: Interbank Memory Moves Without DMA

I was at first disappointed after reading the CP/M Plus documentation because it insists that DMA hardware be used for interbank data movements. My machine has a DMA disk controller, but no way of asking it to perform a memory-to-memory move operation. After thoroughly studying the sample BIOS listings included in the system manual, however, I couldn't think of any reason to doubt that I would be able to find a programmable solution to this problem.

The listing on page 20 shows the MOVE module that I am currently using. It is patterned after Digital Research International's (DRI) suggested routine with only a few small modifications. So far it has proven very reliable and with only two changes should work equally as well in most any Z80 hardware configuration with or without special DMA features.

The first routine, XMOVE, is provided for banked systems that support memory-to-memory transfers over the entire extended address range. Systems with this feature can have their data buffers located in an alternate bank of memory instead of in common memory. By relocating the data buffers away from common memory (application program) you easily can achieve a TPA of 60K or larger.

A call to XMOVE affects only the following MOVE call. Each time you make an interbank transfer, you must first call XMOVE to establish the data source and destination bank numbers before calling MOVE.

The second routine, MOVE, performs the memory-to-memory block move operation. My version is slightly enhanced to allow any size interbank move. The sample routine only permitted 128 bytes to be transferred at a time, which for me would cause excessive overhead. My hardware will not support direct interbank memory moves so I used a combination of Z80 block move instructions to perform the move in segments of 128 bytes each. This required that each memory segment be stored first in a common memory buffer and then moved to the correct destination address after the appropriate memory bank had been switched into context. This entire process is not nearly as complicated as it may sound, and outside of a need for special hardware, it is the most efficient way to implement interbank moves.

The last routine, BANK, is as dangerous as any in CP/M Plus. At entry to this routine the A register contains the number of the CP/M Plus memory bank (0 through 15) to be switched into context. Because I use the older style bank select method for memory bank switching, I first used a translation table to convert the CP/M bank number to one understood by my hardware.

As I said, the MOVE module is not large in size but does perform some of the most important functions in a banked version of CP/M Plus.

Z80ASM: An Assembler Language Development System

Continuing with July's discussion on Z80ASM from SLR Systems (an assembler language development system), the package I received contained a single 8-inch disk and a standard-size, three-ring documentation binder. There were six files on the disk: four executable programs, a sample assembler language source file for testing the package after installation, and a "READ.ME" file containing recent additions or updates to the documentation manual. I refer to Z80ASM as a package because a linkage editor and a librarian are also included. For the purpose of this discussion, I will restrict my comments to the assembler and the configuration utility program.

To prepare Z80ASM for use, you must run the system configuration utility program. During this setup procedure, you must answer a number of questions about the version of CP/M in use and how the assembler's many default parameters are to be set. Some of the questions are detailed enough that it would be wise to first study the configuration section of the manual before you attempt to run the configuration utility.

Many of the default parameters pertain to the format of the printed hardcopy listing. In addition to specifying how many columns across and lines down are to be printed on each page, you can select options to alter the page appearance. One option, for example, allows the line number assigned by the assembler to be printed on the left margin or between the listing of the generated object code and the actual source statement.

Another option that is particularly important to me is the inclusion of the date and time into the page headings. To gather the date and time from the clock, Z80ASM senses the operating system version in use and issues the appropriate BIOS function calls. If the CP/M system in use doesn't automatically support built-in clock functions, you can specify the memory address of a subroutine that can be called to pass along the clock data.

Z80ASM will also initialize the printer with up to eight characters of

user-specified control information before printing starts and again at the listings termination—certainly a very important feature for the narrow carriage printer owner.

The default filename extension for all input, output, and work files can also be configured to meet your standards. You can eliminate the monotonous task of keying the parameters that control the assembly options by specifying them once at configuration time (and then give them no further thought unless there is some special need). Of course, if you need a special option, it is not necessary to rerun the configuration utility. Practically every permanent option can be overridden if you specify an execution time parameter.

Running Z80ASM for the first time is where the fun begins. As I mentioned in the first installment of this review, I was skeptical when told I could expect a sixfold speed improvement. My first assembly, however, ver- accurate and that my fears were unfounded. I'm told by the author that this incredible assembly speed is due largely to making every use of the Z-80's instruction set and to buffering all disk data into 1K segments. The data rates are even further enhanced when you can use the multi-sector function of CP/M Plus.

A full complement of operator commands are available to control the execution of Z80ASM. If you have ever sat blindly staring at your CRT questioning why a program is running so long, or worse yet, whether it is running at all, you will appreciate these features. If at any time it is necessary to prematurely end an assembly, simply typing a CTL-C will abort the job without your needing to resort to more drastic measures such as hitting the reset button. Further, assembly operation may be temporarily suspended if you key CTL-S and later resume by typing CTL-Q.

Runtime control of the console and printer devices is also provided. The console driver may be enabled and disabled (toggled) with a CTL-Z, and the list device driver may be toggled with a CTL-P. Note that both of these commands may be used whether or not their drivers were enabled in the command line. These useful commands let you view just a portion of the assembly without requiring you to insert control statements into the program source code. And in the unlikely event that you may be questioning what Z80ASM is doing at any moment, typing a question mark will display the current filename, line number, and source line that is being assembled.

Errors that creep into your assembly are described by over 30 full text messages. Not only are they printed in the listing, but they are also displayed on the CRT as the assembly runs.

To ensure that no error messages slip by, you will be asked during the configuration process for the number of lines to be displayed on the CRT before the assembler stops and waits for you to command it to continue. A maximum error limit can also be set that, when reached, automatically aborts the assembly. In any event, if the assembly ends with errors present, the CP/M Plus error code in the system control block is set to FF00.

A great deal of an assembler's usefulness comes from the flexibility of the pseudo-ops it recognizes as well as a well-integrated macro language. Again, Z80ASM goes one step beyond what is expected of a good assembler by providing several unique pseudo-ops that allow operator entry of values and the display of text on the CRT at runtime. Of particular interest to me were the string comparison pseudo-ops (all of this in addition to those carried forward from M80 and DRI's RMAC assembler).

In summary, Z80ASM is an extraordinary product that must be considered by anybody who does assembly language programming. Not only does the assembler work as advertised, it is also well supported.

If you want to know more about Z80ASM or the other products offered by SLR Systems, you can contact the company directly at 200 Homewood Drive, Butler, PA. 16001; in Pennsylvania, call (412) 282-0864, other states (800) 833-3061.

DDJ

(Listing begins on page 20)

CP/M Exchange Listing (Text begins on page 18)

```
                .z80
                title   MOVE - bank select & move module for CP/M Plus
                subttl  save areas and equates
                .comment||
                page

        ;
        ;       Global definitions for CP/M Plus
        ;

        false           equ     0
        true            equ     not false
        banked          equ     true
        ;
        ;

                maclib  cpm3glbl.lib

                cseg

        ;
        ;       subroutines for others modules use
        ;

                public ?move,?xmove,?bank
                extrn   ?pmsg,?pdec
        ;
        ;       external areas and subroutines that we may need to use
        ;

                extrn   @cbnk,@dskbf
                extrn   ?bnksl

        ;
        ;       local save areas
        ;

move_stat:      defb    0                       ; flag to show next move is interbank

src_des_bank    equ     $
src_bank:       defb    0                       ; source bank number
des_bank:       defb    0                       ; destination bank number

des_add:        defs    2                       ; destination address for interbank move
src_add:        defs    2                       ; source address for interbank move
mov_cnt:        defs    2                       ; byte count for interbank move

hold_hl:        defw    0                       ; saveareas for callers registers
hold_de:        defw    0
hold_bc:        defw    0

old_stk_save:   defs    2                       ; place to save users stack
                defs    16                      ; place for our local stack
local_stack     equ     $

tc_bank:        defb    0

conv_bank_tbl:  defb    01h,02h,04h,08h,010h,020h,040h,080h
                                                ; conversion table used to translate
                                                ; CP/M's bank to ours

        ;
        ;       local equates
        ;

bank_port       equ     040h                    ; bank selection port address
int_bnk_act     equ     0                       ; flag for interbank move active
```

(Continued on page 22)

```
                subttl  ?XMOVE - interbank move setup subroutine
                page
        ;
        ; set interbank move flag active for next call to ?move subroutine
        ; and save both the destination and source bank numbers
        ;
        ; at entry:
        ;
        ; b = destination bank
        ; c = source bank
        ;

?xmove:

                ld      (src_des_bank),bc       ; save source and destination
                ld      a,(move_stat)           ; get status byte
                set     int_bnk_act,a           ; set interbank move active
                ld      (move_stat),a           ; save status byte
                ret                             ; return to caller

                subttl  ?MOVE - memory to memory move subroutine
                page

?move:

                ld      a,(move_stat)           ; test flag for interbank move
                bit     int_bnk_act,a           ; and go to appropriate routine
                jr      nz,int_bnk_mov          ; *

                ex      de,hl                   ; we are passed source in de and dest in hl
                ldir                            ; use z80 block move instruction
                ex      de,hl                   ; need next addresses in same regs
                ret                             ; return to caller

        ;
        ;       this routine is improved over the DR example
        ;
        ;       interbank moves of greater than 128 bytes can be made with one exception,
        ;       a move cannot span a bank boundary

int_bnk_mov:

                res     int_bnk_act,a           ; set off interbank move flag
                ld      (move_stat),a           ; save status byte

                ld      (old_stk_save),sp       ; save callers stack
                ld      sp,local_stack          ; and load it with a local stack

                ld      (des_add),hl            ; save destination address
                ld      (src_add),de            ; save source address
                ld      (mov_cnt),bc            ; save length of move

int_bnk_mov_a:

                ld      hl,(mov_cnt)            ; restore move length
                ld      a,l                     ; check for zero length move
                or      h                       ; *
                jr      z,int_bnk_mov_ret       ; *
                ld      bc,128                  ; segment length
                xor     a                       ; clear carry flag
                sbc     hl,bc                   ; decrement length
                jr      c,int_bnk_mov_last      ; last move this time
                ld      (mov_cnt),hl            ; save remaining length
                ld      hl,(src_add)            ; restore source address
                ld      de,@dskbf               ; point at intermediate save area
                ld      bc,128                  ; segment length
                ld      a,(src_bank)            ; get source bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (src add),hl            ; save new source address
```

```
                ld      hl,@dskbf               ; point at segment as source
                ld      de,(des_add)            ; restore real destination address
                ld      bc,128                  ; segment length
                ld      a,(des_bank)            ; get segment bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (des_add),de            ; save new destination address
                jr      int_bnk_mov_a           ; loop until complete

int_bnk_mov_last:

                ld      hl,(src_add)            ; restore source address
                ld      de,@dskbf               ; point at intermediate save area
                ld      bc,(mov_cnt)            ; remaining length to move
                ld      a,(src_bank)            ; get source bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (src_add),hl            ; save new source address
                ld      hl,@dskbf               ; point at segment as source
                ld      de,(des_add)            ; restore real destination address
                ld      bc,(mov_cnt)            ; remaining length to move
                ld      a,(des_bank)            ; get destination bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (des_add),de            ; restore ending source address

int_bnk_mov_ret:

                ld      a,(@cbnk)               ; load current bank number
                call    ?bnksl                  ; find physical bank and swap it
                ld      sp,(old_stk_save)       ; restore callers stack pointer
                ret                             ; back to caller

                subttl  ?BANK - select bank subroutine
                page

?bank:

                ld      (hold_hl),hl            ; save callers registers
                ld      (hold_de),de            ; *
                ld      (hold_bc),bc            ; *
                ld      c,a                     ; desired bank number to bc
                ld      b,0                     ; *
                ld      hl,conv_bank_tbl        ; pointer to bank conversion table
                add     hl,bc                   ; create memory pointer to desired byte
                ld      a,(hl)                  ; load bank number
                out     (bank_port),a           ; swap in desired bank
                ld      a,c                     ; restore bank number
                ld      hl,(hold_hl)            ; restore user registers
                ld      de,(hold_de)            ; *
                ld      bc,(hold_bc)            ; *
                ret

                end
```

End Listing

File Maintenance in Forth

by Ray Cadmus

Last year I was asked by a group of ceramics shop owners to develop a computer system to help them run their businesses more efficiently. Since my wife, a shop owner herself, was the spokeswoman for the group, how could I say no? Their wants were few. Keep it cheap, simple and comprehensive—in that order! To keep it cheap I looked to the Commodore 64 computer where a complete system with disk and printer was available for less than $1000 (not the best choice, as it turned out). The need for simplicity dictated that everything must be self-prompting or menu-driven. As for comprehensive—the want list grew . . . .

The group needed the computer to function as an electronic cash register, capturing both financial and inventory control information. Automatic price, product description and customer information lookup would be "nice." In general, they wanted inventory control, mailing list maintainance, class scheduling and full accounting functions including vendor and customer histories. With so much data to handle and so many files to be built and maintained, the small computer had to fill a tall order. This article presents my response to the challenge a simplified means; written in Forth, that builds and performs much of the routine maintainance on many types of data files.

Ray Cadmus, 600 W. Lee, Moberly, MO 65270.

Forth's extensibility makes it ideal for providing a core on which to build more complex structures. This data-management core begs to be extended, and the author shows some data-management tools that can be built from it.

Why Forth?

When I set out to do a simple business application on a Commodore 64, I quickly discovered that BASIC on the C64 is just too limited to really do justice to the power and capabilities of the machine. Forth was readily available; it is fast enough for dynamic graphics and sound routines (although neither is demonstrated in this project); and it is extensible, so it can grow more naturally to handle the features of whatever computer uses it—besides, I had been looking for an excuse to learn Forth anyway.

File Maintenance and Forth

One of the more common functions of a small computer is the storing and retrieving of data. Whether you call this data management or file maintenance, it normally means writing a series of programs to build and maintain individual data files. The Forth screens in the listing on (page 26) present a series of routines or "words" that allow the very simple description of a data file and its associated screen format. The routines provide capabilities for adding new records and for changing existing ones. You may change or retain data within a record on a field-by-field basis. Once you have established the file maintenance routines as a part of the Forth system, file handling is a matter merely of your defining a table word, setting a starting BLOCK and re-

cord length, then stepping into the maintenance MENU for the rest of the operations.

Before we look at the details of the program, it may be useful for me to describe a few of the terms I'll be using and to examine how Forth's handling of data differs slightly from most conventional programming systems.

For most people, the smallest practical unit of data is the character or "byte" of data. We combine these characters into logical, meaningful groups to make up a data element or "field" of data. This field may be numeric, as in a quantity or price field, or it may be alphabetic as in a description field. We can combine related fields into a "record." For example—an ITEM RECORD may contain an ITEM NUMBER field, a DESCRIPTION field, a PRICE field and a QUANTITY-ON-HAND field. This grouping makes up a "logical record." A "file" is a collection of logical records stored on a device outside the computer—usually a disk drive.

Both records and fields may be either variable-length or fixed length. Variable length units are usually separated by a common delimiter of some sort such as a comma, but fixed-length units are not; for example, our item record:

123,BIG BOX,19.95,15 (variable)
00123BIG BOX 019950015 (fixed)

A variable-length item takes up less room but a fixed-length record is more predictable. For instance, because we know how long each record is, we can calculate where any record will be by multiplying the number of preceding records by the record length. This is the basis for random access files; that is, we can go directly to the desired record without searching through each record from the beginning of a file.

How is Forth different? Forth organizes and stores its data as 1024 character "physical records" known as BLOCKS or SCREENS, stored by block number. We may store our "logical records" into a Forth BLOCK.

The Forth package used originally in the development of this package was the Commodore 64 Forth by Performance Micro Products (PMP). I have since converted the file management system to run under Laboratory Microsystems (LM) Z80 Forth. The ease of this conversion is a testimony to the portability of Forth. PMP Forth and LM Forth both are based on FIG Forth, with a few extensions and a few differences. The major difference (and the reason that I used the LM version for the enclosed examples) is that PMP Forth uses a 24 × 40 screen format rather than the standard 16 × 64 format of LM Forth. I encountered several minor differences: NUMBER in LM Forth requires a trailing blank to terminate while PMP Forth accepts either a blank or a NUL; and PMP Forth uses ?DUP where LM Forth uses –DUP.

Screen Descriptions

Screen 105 describes the terminal control as used by my Radio Shack TRS-80 Model II, and screen 106 defines a simple, relative record file access system. The word RECORD takes the record number off the stack, reads the data block if necessary, then sets up a pointer to the record on the block. You can change RECORD to use the native operating system's file system instead of the Forth block structure if you desire. At the very minimum, you should change it to check for a maximum block value so that your file doesn't overwrite something it shouldn't. You may use record 0 to keep miscellaneous data about the file. Currently, the first two bytes in record 0 contain the number of records in the file. You should set this count to 0 for a new file. For example:

(0 BBLOCK @ block !)

Screens 107 and 108 contain some of the words used by the data entry portion of the system. ?DIGIT returns a true or false flag showing when the top stack entry is an ASCII digit. ?NUMBER determines when the text string pointed to by the top stack entry is convertible to a number. Using these words ensures that the program won't "blow up" from unusable garbage. ASK prompts for a yes/no and returns a true flag when the response is Y. LEN returns the length of a string. WAIT is a pause word used to keep data on the screen until you have viewed it. INPUT accepts a number from the terminal, converts it to internal form, and leaves it on the stack as a double number.

Screens 109 – 118 are the file maintenance system (FMS) words, and screens 119 and 120 are a couple of sample applications using the FMS. The key to the FMS is the word PBUILD, which builds a page description table. It is called with the top of stack containing the number of fields to be displayed on the data entry screen. Following the word PBUILD are the field description entries themselves (see screens 119 and 120 for examples).

The set of entries for each line of the PBUILD table are:

(1) The line on the screen to display the prompt
(2) The screen column for the prompt display
(3) The line on the screen to display entered data
(4) The screen column for data entry
(5) The length of data to enter
(6) The data type code where:
    0 – no data entry, display only
    1 – numeric single byte
    2 – numeric single word
    3 – numeric single word; treat as a dollar amount, display as nnnn.nn (2 decimals)
    4 – numeric double word
    5 – numeric double word; treat as a dollar amount
    9 – alphanumeric text
(7) The location in the record of the data (base 0)
(8) The prompt text delimited by trailing quotes

That's it! This simple table definition sets up everything for data entry, display, and update. Note that the type code determines how much room a numeric field takes up in the data record.

Screen 111 contains the FFIND routine that finds the location of the field description data in the table. FDISPLAY displays the prompt for a given field, and PDISPLAY displays the entire screen of prompts.

Screen 112 describes FDATA and ?FDATA, which determine where the data is in the record and what type it is. ?FDATA is used only by FDATA.

FGET in screen 113 uses FTEXT and FNUM to accept input data and place it into the proper record location. FPUT on screen 114 takes a field's data from the record and places it on the screen. PPUT and PGET on screen 115 force the display and entry, respectively, of a whole screen and record full of data.

The words in screens 109 – 115,

which form the guts of file maintenance, could also be used as screen and record managers in other programs. The rest of the words on screens 116 – 118 use the file and screen words to develop a simple FMS. You could easily enhance these routines to add record indexing and report generation for a full-fledged data base manager.

Screens 119 and 120 are examples of two applications using the simple FMS. The first is a standard name and address file system. The second maintains a mold catalog for a ceramics shop.

Summary

The steps required to build and use a file with this package are:

(1) Draw up the desired screen layout, and mark row and column positions for each field.

(2) Define the description table using PBUILD.

(3) Pick a range of blocks for the file, and set the record count in the first block to 0.

(4) Define the top-level application word to:

- Call IREC to initialize the starting block and record length
- Set up the page table pointer with PINIT
- BEGIN PMENU UNTIL

(5) Start the application by invoking the top-level word.

Once started, the maintenance function is menu-driven. Selecting the Add function adds a new record as the next sequential record number. You should modify List for the application; as it stands, it displays a record number and the first 20 characters of each record. Change and Display both select by record number. You could easily change Display to do a search on a specific field. Change displays the selected record as it currently exists and allows changes on a field-by-field basis. To step past a field without changing it, press Return. To change a field, type over it.

I hope this little system and its supporting routines prove useful. Keep in mind that this package, while useful as it stands, is really just the demonstration of a concept and is begging for expansion. Items that should be added are a record delete function, keyword indexing, and a full screen editor to assist in building the PBUILD table. In the meantime, it has served its primary purpose of helping me learn a little bit about Forth.

A Postscript

I commented earlier that perhaps the Commodore 64 was not the best choice for this project. I don't want to leave you with the wrong impression. The C64 is a great little computer for the money; its weakness for this business application is the slow speed of disk drives. Disk copies take so long to create that they would probably be ignored by the casual or inexperienced user; a lack of backup files might eventually prove disastrous.

DDJ

File Maintainance Listing (Text begins on page 24)

```
Screen # 105
  0 ( SCREEN CONTROL - XY & CLS FOR RADIO SHACK II P&T CP/M)
  1
  2 : XY     ( ROW COL ---   )
  3          SWAP 27 EMIT 89 EMIT
  4          0 MAX 23 MIN 32 + EMIT 0 MAX 79 MIN 32 + EMIT ;
  5
  6 : CLS    ( HOME AND CLEAR SCREEN)
  7          12 EMIT ;
  8
  9 -->
 10
 11
 12
 13
 14
 15

Screen # 106
  0 ( RECORD ACCESS )
  1
  2 0 VARIABLE RLOC          0 VARIABLE RLEN          0 VARIABLE RNO
  3 0 VARIABLE REC/BLK       0 VARIABLE BBLOCK
  4
  5 ( BBLOCK IS STARTING BLOCK FOR THE FILE )
  6 ( RLEN IS LOGICAL RECORD LENGTH )
  7
  8 : IREC   ( BBLOCK RLEN ---       INITIALIZE RECORD POINTERS)
```

```
 9          RLEN ! BBLOCK ! ;        ( BBLOCK IS START OF FILE)
10
11 : RECORD ( RNO --- ADDR     RETURNS RECORD ADDRESS )
12          RNO !    1024 RLEN @ / REC/BLK !
13          RNO @ REC/BLK @ /MOD     ( POSITION IN BLOCK)
14          BBLOCK @ + BLOCK SWAP RLEN @ * + DUP RLOC ! ;
15 -->

Screen # 107
 0 ( NUMERIC CHECK ROUTINES - ?DIGIT ?NUMBER  )
 1 0 VARIABLE DFLAG          ( DIGIT FLAG)
 2
 3 : ?DIGIT          ( N --- F     IS CHARACTER A DIGIT? )
 4          DUP 47 > SWAP 58 < AND ;
 5
 6 : ?NUMBER         ( LOC --- F )
 7          0 DFLAG !          DUP 11 + SWAP DO
 8                    I C@ ?DIGIT IF 1 DFLAG !
 9                               ELSE
10                    I C@ 32 = I C@ 0= OR
11                               IF LEAVE ELSE I C@ 46 = 0=
12                               IF 0 DFLAG ! LEAVE
13                    THEN THEN THEN LOOP
14          DFLAG @ ;
15 -->

Screen # 108
 0 ( MISC ROUTINES           ASK  LEN  WAIT )
 1
 2 : ASK             ( --- F          PROMPT Y/N )
 3      '    ." (Y/N) " KEY 78 = 0= ;        ( TRUE IF YES )
 4
 5 : LEN    ( ADDR --- COUNT )   ( RETURN LENGTH OF STRING )
 6      255 0 DO DUP I + C@ 0= IF I LEAVE THEN
 7          LOOP SWAP DROP ;
 8
 9 : WAIT   ( --- )   ( WAIT FOR KEYPRESS )
10      CR    CR    ." ANY KEY TO CONTINUE"   KEY DROP ;
11
12 : INPUT      ( NUMBER INPUT )
13       PAD 11 EXPECT PAD LEN PAD + 32 SWAP C! PAD 1 - NUMBER ;
14 -->
15

Screen # 109
 0 ( FILE MAINT - PINIT )
 1 0 VARIABLE PHIGH          ( HIGH ENTRY ON THE FILE )
 2 0 VARIABLE FNO            ( FIELD NO                )
 3 0 VARIABLE FLOC           ( FIELD LOCATION IN RECORD      )
 4 0 VARIABLE PDONE          ( PAGE COMPLETE FLAG    )
 5 0 VARIABLE PLOC           ( PAGE TABLE LOCATION   )
 6 0 VARIABLE PMAX           ( NUMBER OF FIELDS ON THIS PAGE )
 7
 8 ( INITIALIZE THE PAGE <SCREEN> PROCESSING WITH THE       )
 9 ( LOCATION OF THE DESCRIPTION TABLE BUILT WITH PBUILD    )
10
11 : PINIT          ( TBLLOC --- )
12          DUP PLOC !        ( REMEMBER WHERE THE TABLE IS    )
13          C@ PMAX C!        ( SAVE MAX FIELD COUNT           )
14          0 PDONE !         0 FNO ! ;        ( CLEAR FLAGS   )
15 -->
```

(Continued on page 30)

File Maintainance Listing (Listing Continued, text begins on page 24)

```
Screen # 110
  0 ( FILE MAINT - PBUILD )
  1
  2 ( THIS BUILDS THE FILE & SCREEN DESCRIPTION THAT CONTROLS )
  3 ( THE WHOLE PROCESS  - ENTER WITH NUMBER OF FIELDS ON STACK )
  4
  5 : PBUILD <BUILDS DUP  C, 0 DO      ( NO_OF_FLDS --- )
  6          32 WORD HERE NUMBER DROP C,  ( ROW TO DISPLAY PROMPT )
  7          32 WORD HERE NUMBER DROP C,  ( COLUMN  "    "    " )
  8          32 WORD HERE NUMBER DROP C,  ( ROW FOR INPUT DATA )
  9          32 WORD HERE NUMBER DR0P C,  ( COLUMN "      "     )
 10          32 WORD HERE NUMBER DROP C,  ( INPUT DATA LENGTH )
 11          32 WORD HERE NUMBER DROP C,  ( INPUT DATA TYPE )
 12          32 WORD HERE NUMBER DROP C,  ( DATA POSITION IN REC )
 13          34 WORD HERE C@ 1+ ALLOT     ( PROMPT TEXT TO DISPLAY )
 14   LOOP DOES> ;
 15 -->

Screen # 111
  0 ( FILE MAINT - FFIND  FDISPLAY  PDISPLAY          )
  1 : FFIND ( FNO --- FLOC          FIND FIELD DATA IN TABLE )
  2          PLOC @ 1+ FLOC ' -DUP IF
  3                   0 DO
  4                   FLOC @ 7 + DUP C@ + 1+ FLOC !
  5                   LOOP
  6          THEN FLOC @ ;
  7 : FDISPLAY ( FLOC --- )         ( DISPLAY FIELD'S PROMPT )
  8          DUP DUP C@ SWAP 1+ C@ XY 7 + COUNT TYPE
  9          FLOC @ 2 + C@ FLOC @ 3 + C@ XY FLOC @ 4 + C@
 10          0 DO 95 EMIT LOOP ;
 11 : PDISPLAY ( --- )  ( DISPLAY FULL PAGE OF PROMPTS )
 12          CLS PMAX C@ 0 DO
 13                   I FFIND FDISPLAY
 14          LOOP ;
 15 -->

Screen # 112
  0 ( FILE MAINT - ?FDATA   FDATA )
  1 ( ?FDATA CHECKS TO SEE IF FIELD GETS DATA - COULD BE )
  2 ( DISPLAY ONLY )
  3
  4 : ?FDATA ( FLOC --- FLOC TRUE  or FLOC --- FALSE )
  5          DUP 4 + C@ 0= IF DROP 0 ELSE 1 THEN ;
  6
  7 : FDATA  ( FNO --- LOC LENGTH TYPE or  --- 0 if non-data )
  8          FFIND ?FDATA IF
  9                   DUP 2+ C@ SWAP 3 + C@ XY ( POSITION CURSOR)
 10                   RLOC @ FLOC @ 6 + C@ +  ( FIELD LOCATION )
 11                   FLOC @ 4 + C@           ( FIELD LENGTH )
 12                   FLOC @ 5 + C@           ( FIELD TYPE )
 13          ELSE     0       ( NOT A DATA FIELD )
 14   THEN ;
 15 -->


Screen # 113
  0 ( FILE MAINT - FGET     FTEXT   FNUM )
  1 0 VARIABLE FTYPE
  2 : FTEXT ( LOC OLEN PLEN --- ) ( GET TEXT TYPE DATA )
  3          ROT ROT 2DUP BL FILL DROP
  4          PAD SWAP ROT CMOVE ;
  5 : FNUM 2DROP PAD DUP 11 + SWAP DO I C@ 0= IF 32 I C! THEN LOOP
```

```
 6       PAD ?NUMBER IF  PAD 1- NUMBER ELSE 0 0 THEN ;
 7 : FGET  FDATA -DUP IF   ( FNO --- )
 8    FTYPE ! PAD OVER EXPECT PAD LEN  -DUP IF
 9                  FTYPE @ 9 = IF FTEXT THEN
10                  FTYPE @ 1 = IF FNUM DROP SWAP C! THEN
11                  FTYPE @ 2 = IF FNUM DROP SWAP ! THEN
12                  FTYPE @ 3 = IF FNUM DROP SWAP ! THEN
13                  FTYPE @ 4 = IF FNUM ROT 2! THEN
14                  FTYPE @ 5 = IF FNUM ROT 2! THEN
15         ELSE 2DROP THEN THEN ;                    -->

Screen # 114
 0 ( FILE MAINT - PUT DATA TO SCREEN  -  FPUT   .$   )
 1 : .$ SWAP OVER DABS     ( DISPLAY MONEY FIELD )
 2         <# # # 46 HOLD #S SIGN #> TYPE SPACE ;
 3
 4 : FPUT ( FNO --- )      ( DISPLAY DATA FIELD )
 5         FDATA -DUP IF FTYPE !
 6                  FTYPE @ 9 = IF TYPE THEN
 7                  FTYPE @ 1 = IF SWAP C@ SWAP .R THEN ( CHAR)
 8                  FTYPE @ 2 = IF SWAP @ SWAP .R THEN ( SINGLE )
 9                  FTYPE @ 3 = IF DROP @ 0 .$ THEN ( DOLLARS )
10                  FTYPE @ 4 = IF SWAP 2@ SWAP D.R THEN
11                  FTYPE @ 5 = IF DROP @ .$ THEN  ( D$'S )
12         THEN ;
13
14 -->
15

Screen # 115
 0 ( FILE MAINT -  PGET    PPUT )
 1 ( SET RLOC <RECORD LOCATION> PRIOR TO CALLING EITHER OF )
 2 ( THESE WORDS )
 3
 4 : PGET ( --- )  ( GET ALL THE ITEMS ON A PAGE )
 5         PLOC @ C@       ( GET THE FIELD COUNT )
 6            0 DO I FGET LOOP ;
 7
 8 : PPUT ( --- )   ( DISPLAY ALL ITEMS FROM A RECORD )
 9         PLOC @ C@       ( # FIELDS )
10           0 DO I FPUT LOOP ;
11
12 -->
13
14
15

Screen # 116
 0 ( FILE MAINT - PNEXT    PADD    PLIST  )
 1
 2 : PNEXT          ( NEXT RECORD POSITION TO ADD )
 3         PHIGH @ 1 + DUP PHIGH ! RECORD ;
 4
 5 : PADD           ( ADD OR CHANGE A RECORD )
 6         PDISPLAY PGET 0 ;
 7
 8 : PLIST          ( LIST SELECTED PORTION OF RECORDS )
 9         CLS
10         PHIGH @ 1 + 1 DO
11                  I 3 .R SPACE
12                  I RECORD 20 TYPE CR  ( NAMES FOR NA TEST )
13                  I 23 MOD 0= IF WAIT CLS THEN
14         LOOP WAIT 0 ;
15 -->
```

(Continued on next page)

File Maintainance Listing (Listing Continued, text begins on page 24)

```
Screen # 117
  0 ( FILE MAINT  -  PSELECT  PEND )
  1
  2 : PSELECT         ( PICK A RECORD FOR FURTHER ACTION )
  3          CLS CR ." ENTER RECORD NO.  "
  4          INPUT DROP
  5          DUP PHIGH @ > IF ." NO SUCH RECORD" DROP 0
  6                    ELSE    RECORD DROP 1 THEN ;
  7
  8 : PEND            ( TERMINATE THIS TASK )
  9          CR ." SURE " ASK IF
 10                    PHIGH @ 0 RECORD ! UPDATE
 11                    CLS ." SAVING FILES" CR
 12                    FLUSH ." BYE" CR 1
 13          ELSE 0 THEN ;
 14
 15 -->
```

```
Screen # 118
  0 ( FILE MAINT - PMENU )
  1 : PMENU CLS 6 10 XY ." FILE MAINTAINANCE"
  2          10 1 XY ." 1 - ADD" CR
  3          12 1 XY ." 2 - LIST" CR
  4          14 1 XY ." 3 - DISPLAY A RECORD" CR
  5          16 1 XY ." 4 - CHANGE A RECORD" CR
  6          18 1 XY ." 9 - QUIT" CR
  7          20 10 XY ." SELECT ONE OF ABOVE " KEY
  8    DUP 49 = IF DROP 0 PNEXT PADD
  9    ELSE DUP 50 = IF DROP PLIST
 10    ELSE DUP 51 = IF DROP PSELECT IF PDISPLAY PPUT WAIT 0 THEN
 11    ELSE DUP 52 = IF DROP PSELECT IF PDISPLAY PPUT PGET 0 THEN
 12    ELSE DUP 57 = IF DROP PEND
 13    ELSE DROP 0
 14    THEN THEN THEN THEN THEN ;
 15
```

```
Screen # 119
  0 ( FILE MAINT TEST - NAME AND ADDRESS APPLICATION )
  1
  2 5 PBUILD NAPAGE
  3 6   10  0   0   0   0   00 MAIL LIST"
  4 10  1   10  15  20  9   00 NAME......."
  5 12  1   12  15  20  9   20 ADDRESS...."
  6 14  1   14  15  20  9   40 CITY STATE."
  7 16  1   16  15  5   9   60 ZIP CODE..."
  8
  9 : NA    ( TASK CALL - REMEMBER TO SET 0 RECORDS AT BLOCK )
 10     130 65 IREC      ( SET START BLOCK & RECORD LENGTH )
 11     0 RECORD @ PHIGH !  NAPAGE PINIT
 12     BEGIN PMENU UNTIL ;
 13
 14
 15
```

```
Screen # 120
 0 ( FILE MAINT SAMPLE 2 )
 1
 2 8 PBUILD M.PAGE
 3 3   5   0   0   0   0   0 MOLD FILE MAINT"
 4 5   1   5   15  5   9   0 VENDOR"
 5 7   1   7   15  10  9   5 VENDOR NO"
 6 9   1   9   15  20  9   15 CATAGORY"
 7 11  1   11  15  24  9   35 DESCRIPTION"
 8 13  1   13  15  6   3   59 G PRICE"
 9 15  1   15  15  6   3   61 COST"
10 17  1   17  15  6   3   63 RETAIL"
11
12 : MOLD
13      131 80 IREC 0 RECORD @ PHIGH !
14     M.PAGE PINIT BEGIN PMENU UNTIL ;
15
```

End Listing

Forth and the Fast Fourier Transform

by Joe Barnhart

Long the province of the scientist and the engineer, the Fast Fourier Transform (or FFT) has found its way into a fantastic number of applications. Heat transfer, medical imaging, and sales data analysis are just a few of the FFT's many capabilities. In this article, I use the FFT to analyze stock market data and detect cyclic trends.

This FFT package is written in the Forth language. Novices to Forth often wonder how any serious work can be done without floating-point numbers. Forth programmers can benefit from seeing how the FFT was converted from an algorithm that is usually performed with floating-point numbers into one that is suitable for Forth's fixed-point math. Not surprisingly, the fixed-point version will run rings around most floating-point versions in terms of execution speed.

This program was written with transportability in mind. Because this program is written in the new Forth-83 Standard version, it should be transportable to many different systems. I used a Forth system from Laboratory Microsystems, Inc. (LMI), that adheres to the new standard and runs under the CP/M-68K operating system. My hardware is a Sage II with an 8-MHz Motorola 68000 processor, but the same code runs without modification on LMI's IBM PC/Forth and Z80 Forth (also Forth-83 Standard).

Those accustomed to using floating-point arithmetic often bemoan Forth's fixed-point orientation. It's remarkable, though, how much you can do—and how efficiently you can do it—using only fixed-point math.

Joe Barnhart, 522 Talbot Avenue, Albany, CA 94706.

A Little Theory

What exactly does the Fourier Transform do? Put simply, it relates time to frequency. For example, imagine a man beating two rocks together at a steady rate. If you graph the sound made by the rocks versus time, you get the curve shown in Figure 1 (page 36). Every fourth point shows the sound made as the rocks came together. On the other hand, you might represent the event by the graph in Figure 2 (page 36). In this graph, you can see that the axes are labeled in terms of sound versus frequency, and the only response is at 0.25 Hz (or 0.25 beats/second). These two graphs are equivalent ways of looking at the same process.

As a more realistic example, consider the tone source for an electronic music synthesizer. It can produce many different musical timbres by altering the shape of the tone waveform. Figure 3 (page 36) shows a square wave. The sound that the wave represents is harsh, sharp, and buzzing because it is composed of many high frequencies. If you look at the square wave with a tool called a spectrum analyzer, you see the pattern in Figure 4 (page 36). The square wave is composed of many different pure sine waves. The Fourier Transform of the square wave yields exactly the same pattern as does the spectrum analyzer.

The Discrete Fourier Transform (DFT) is a simple extension of the Fourier Transform that handles data that is represented as points instead of continuous curves (Figure 5, page 36). The Fast Fourier Transform (or FFT)

is a quick method of computing a Discrete Fourier Transform.

To gain speed in computing a Fourier Transform, the FFT algorithm requires that the number of input samples (data points) be an integral power of 2 (e.g., 64, 128, 256). Also, the number of time samples (input to the FFT) and the number of frequency samples (output from the FFT) are the same. These restrictions are a small price to pay for the tremendous increase in speed of the FFT over the DFT algorithm.

How much faster is the FFT? In theory, the time required to calculate a DFT is proportional to N^2, where N is the number of pointers in the input data. The FFT requires time proportional to only $N*\log_2 N$. For example, say a DFT takes one second to calculate a 128-point transform. The FFT of the same data on the same computer can be calculated in only 55 milliseconds!

For the technically inclined, the FFT algorithm presented here is a radix-2, decimation-in-time version. This program takes a complex input array and returns a complex output array—I haven't taken advantage of special symmetry conditions to make the algorithm run faster. On the other hand, this is the most general use of FFT algorithms because it makes no assumptions about the input data.

The Whole Banana

For clarity and modularity, this program is broken in to four distinct modules: a complex math package, a one-dimensional array package, the FFT algorithm itself, and the demonstration application.

Complex Numbers

To implement an FFT program, the computer must be able to use complex - 1. The value of a complex number is derived from the square root of negative one. I won't dwell on the theory of complex numbers here: instead, I'll discuss this implementation of a complex-number package in Forth.

Complex numbers occupy twice as much space as normal integers in Forth. Each complex number is composed of two distinct parts, called the "real" and the "imaginary" parts of the number. In my implementation, the real part is always on the top of the stack and the imaginary part underneath (Figure 6, page 36). Routines are defined to add, subtract, divide, and multiply complex numbers (X+, X–, X*, X/). Other functions unique to complex numbers are magnitude (|X|) and conjugation (X').

The complex arithmetic functions X* and X/ operate on scaled numbers. For instance, I often scale numbers so that 10000 represents the number 1.0000. Whenever two scaled numbers are multiplied, the result must be divided by the scale factor. Similarly, after dividing two scaled numbers, the result must be multiplied by the scale factor. To increase accuracy, intermediate products are maintained in double precision, as in the Forth word */. Addition and subtraction of scaled numbers is identical to the same operations performed on numbers without scaling.

One-Dimensional Array

A one-dimensional array is often referred to as a vector in math and engineering circles. This vector definition is a general-purpose one and can serve many programs other than this FFT. The vector data structure (Figure 7, page 37) contains the number of vector elements, the size of each element (in bytes), the scale factor of the vector, and the number of bytes of storage used by the data area of the vector.

Independence from machine word size is always beneficial when moving an application from one computer to another. This array is indexed by element number rather than by byte offset. A complex number requires four bytes (in a 16-bit Forth system) and an integer 2, but the seventh element of a complex array is accessed in exactly the same way as the seventh element of an integer array. Because the size of each entry is stored with the array, it is a simple matter for the indexing word, [I], to determine the byte offset from the element number.

FFT Routine

This algorithm was taken from a Fortran program (reference 2, page 40). It is a simple, direct method of calculating the Fourier Transform of discrete points. The program consists of two parts: a bit reversal and the FFT "kernel" itself.

This program can compute both the FFT and the Inverse FFT (IFFT). The intended direction of the transform is held in the variable DIR (true = inverse transform). The address of the data array and the number of elements in the array are held by variables, because they are referred to at all levels of nesting.

Bit reversal reorders the vector in preparation for the computation of the transform. This "shuffling" of the data is one reason why the FFT is so much quicker than the DFT. Figure 8 (page 37) shows a small data array and how it is reordered by the routine BIT-REVERSE.

After ordering of the data, the transform itself is computed as a loop within a loop. The inner loop (INNER-LOOP) is factored out of the outer loop (FFT-KERNEL) for readability. At each point in the inner loop, a computation nicknamed "butterfly" is performed. Looking at the diagram of the butterfly computation (Figure 9, page 37), you can see how the name was chosen. Notice that direction of the FFT is tested in the routine BFLY. One of the few differences between a FFT and its inverse is that the FFT is scaled (divided) by the number of points in the data array.

To free myself from the tyranny of trignometric functions, I created a table, WTAB, that contains the sines and cosines needed for FFT computations. WTAB is in the same form as a complex vector, so you can access it with the same words used for vector operations (especially [I]). For FFT's, only the sine and cosine for angles of pi/N are needed, where N is the number of data points. This table can handle FFT's of up to 128 points and is easily extended for larger FFT's.

The Application

The hardest part of having an FFT program is deciding what to use it on. With so many potential applications, it's hard to settle on one for demonstration purposes. I originally wrote this package for use with image processing applications, but it is so useful I'm finding new applications daily—such as looking at stock market trends.

Few of us doubt that stocks move in cyclical patterns. In screen 35 (page 49) I've tabulated the price of IBM common stock by week for all of 1983. The price is normalized to 100—for in-

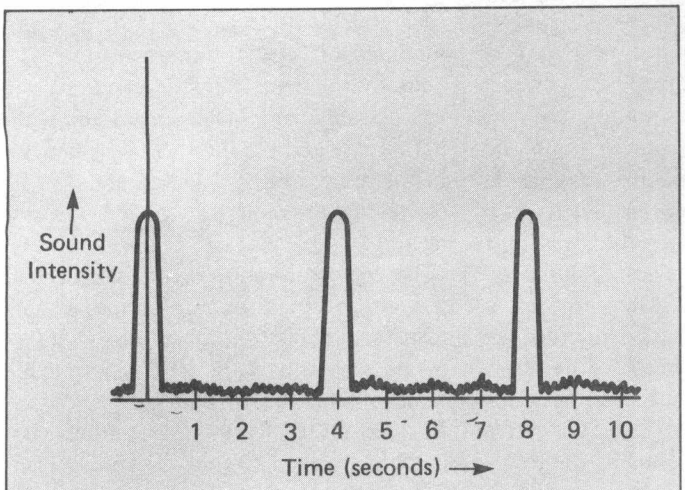


Figure 1 A sound is made every 4 seconds.

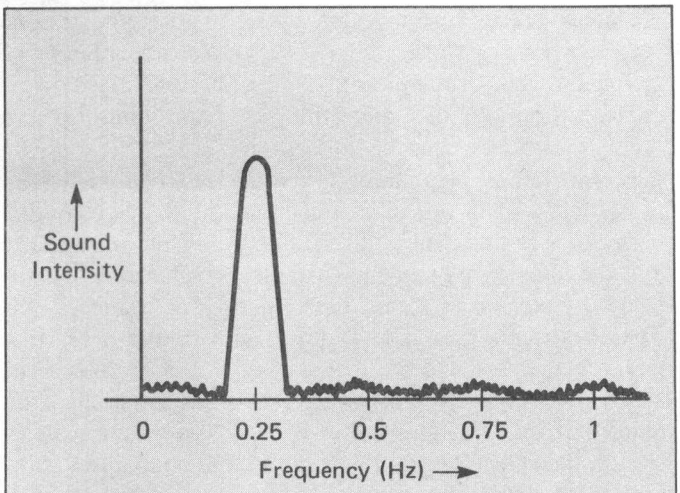


Figure 2 Another way of graphing the same sound.

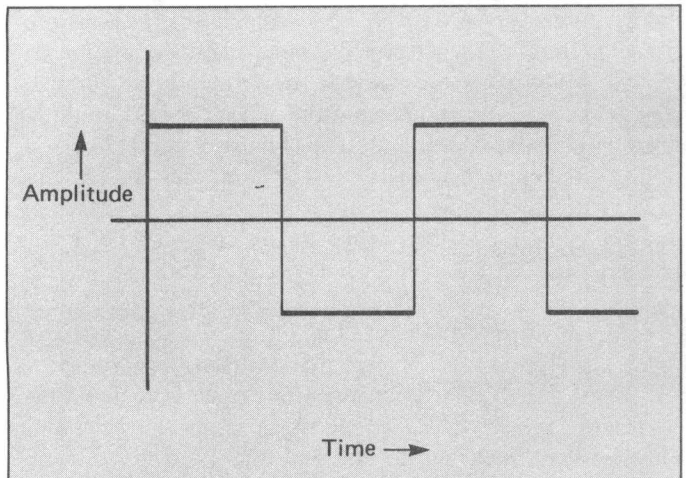


Figure 3 A "square" wave.

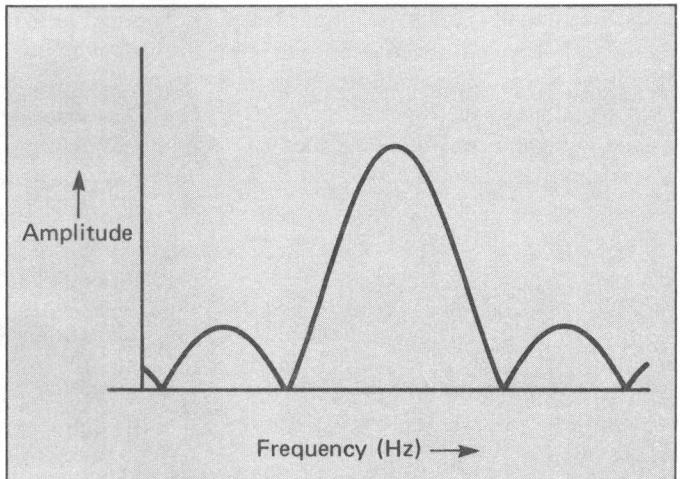


Figure 4 The frequency content of the square wave.

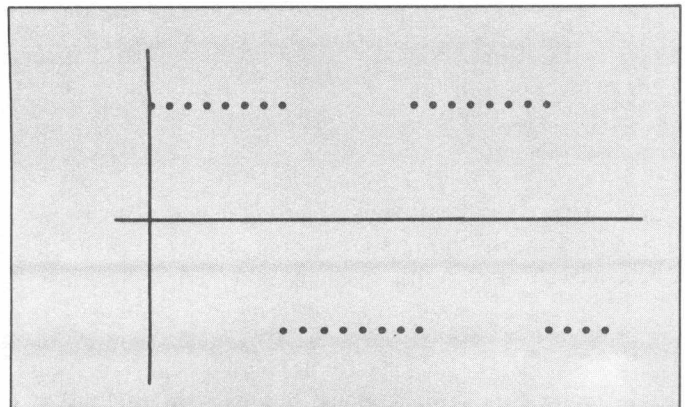


Figure 5 "Discrete" data in the form of a square wave.

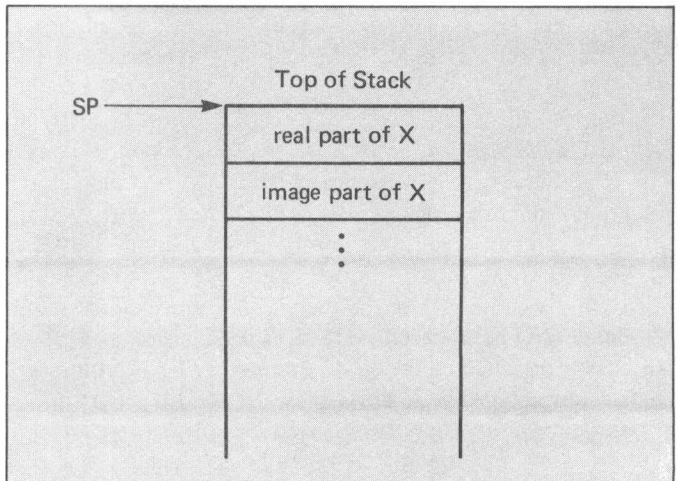


Figure 6 Complex number representation on the stack.

stance, a price of $117 and 7/8 is entered as 11788 ($117.88). Just looking at the raw data, we can see some evidence of cycles (Figure 10, page 38). Some cycles in this data are masked by noise—hidden from the casual observer.

A few notes about the data used in this example are in order. Notice that I have only 52 data points to transform. The FFT only works with a number that is an integral power of 2 (such as 64). To use the FFT in this case, I've filled the array with zeros up to 64 points. When using the FFT on this data, some phantom responses can creep in because of the zeros. I've reduced the errors by employing a special technique called windowing the data—a process that reduces the sharpness of the data and hence the phantom responses (reference 1, page 40).

Performing a Fourier Transform on this stock data, we get a plot of all cycles in the data and their relative sizes (Figure 11, page 40). Even after disregarding normal endpoint effects (indices 1 . . 9 and 10 . . 14 on Figure 11), a few cycles are left. One curious cycle is at index 24, which corresponds to an event that repeats every 20 months or so. Along with their knowledge of the company, some stock chartists use the FFT to help forecast the future price of a stock, given its history.

Other Uses

My original use for this FFT package involved the field of image processing. I used this one-dimensional FFT to build a two-dimensional FFT and performed filtering and restoration on digitized images.

Other uses for this transform abound. If you have a microphone and can digitize its output, you can use the FFT to estimate the frequency response of your stereo system. Feed your stereo with a white noise source and sample the output from the microphone at intervals of 25 microseconds or so. The FFT of the microphone's output indicates the frequency response of your stereo.

An exciting use of the FFT is in the field of shape recognition. Using a touch tablet or digitizer, draw a particular shape (such as a square) over and over again, in various sizes and orientations. Take the (x, y) pairs from the touch tablet and run them through the FFT. From the output of the FFT, coefficients can be calculated (called "Fourier descriptors") that are unique for each shape processed. Recognition of shapes and characters is much easier with these descriptors because they are invariant to the size, rotation, and position of the shape on the tablet.

References

1. *Theory and Application of Digital Signal Processing*, Lawrence Rabiner and Bernard Gold, Prentice Hall, Englewood Cliffs, NJ. A good

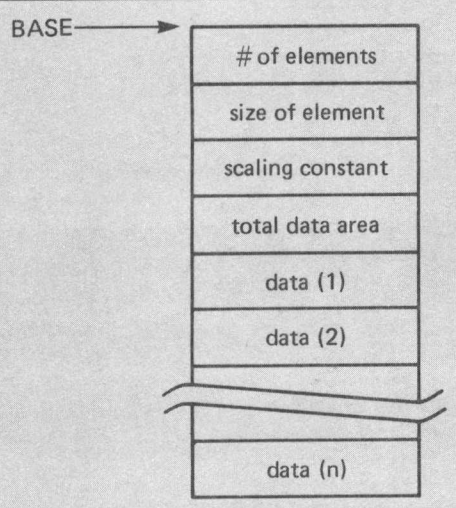


Figure 7
Vector data structure.

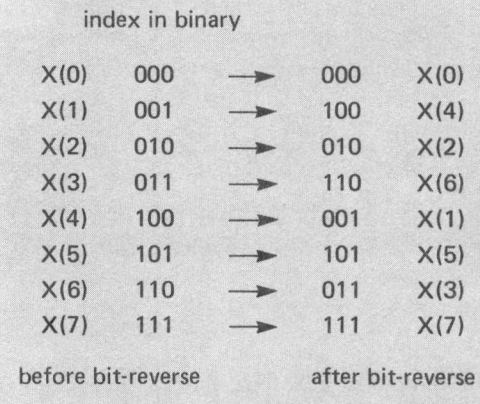


Figure 8
Bit-reversal of a small array.

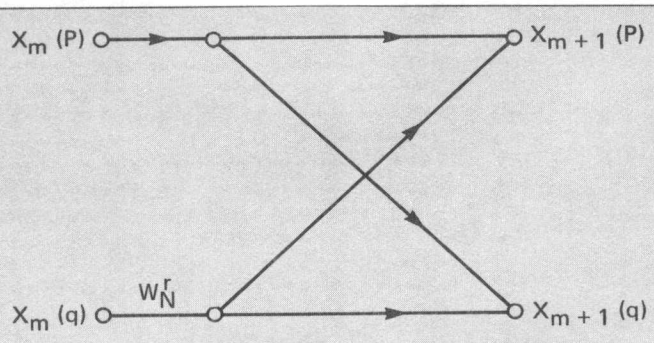


Figure 9 Flow graph of a "butterfly" computation.
(Source: *Digital Signal Processing,* Oppenheim & Schaeffer, Prentice-Hall, pg. 296, fig. 6.9)

second book on the FFT and its uses. You should be familiar with the rudiments of the FFT before tackling this book.

2. *Digital Image Processing*, Rafael Gonzales and Paul Wintz, Addison-Wesley, Reading, MA. Excellent book on all aspects of image processing, including the two-dimensional Fourier Transform and the material on Fourier descriptors.
3. *The Fourier Transform and its Applications*, Ronald Bracewell, McGraw-Hill, New York, NY. A good first text on the Fourier Transform, this book requires first-year calculus for full understanding.

Glossary

These entries are stack diagrams and definitions of the Forth words defined in the FFT program. I've pointed out non-Forth-83 words where I've used them. Stack contents are shown before execution (left side) and after execution (right side). The top of the stack is on the right side of each list. The words are listed in order of their definition.

D2* (dbl dbl --- dbl)
Multiply a double-precision number by 2. Used by the square-root routine.

EASY-BITS,
2'S-BIT,
1'S-BIT
Captive definitions used only by the double-precision square-root routine.

SQRT (dbl --- sgl)
Take the square root of a double-precision number and leave the single-precision result on the stack. This word uses the double-precison extension word DU< (which is Forth-83 but may not be on your system). It can be defined as:

```
: DU< ROT SWAP 2DUP U<
        IF 2DROP 2DROP -1
        ELSE = >R  U<  R>
        AND
        THEN ;
```

SQR (sgl --- dbl)
Square the single-precision number *and leave* the double-precision result on the stack. This definition uses the word M*, which *is not* Forth-83 Standard. M* takes two single-precision arguments and leaves a signed, double-precision product on the stack. An equivalent Forth-83 definition is:

```
: M* OVER OVER XOR >R
        ABS SWAP ABS UM*
     R> 0< IF DNEGATE
     THEN ;
```

X@ (addr --- cmplx)
Fetch a complex number from address and leave it on the stack.

X! (cmplx addr ---)
Store a complex number at address.

XVARIABLE (---)
Create a named entity that, when executed, leaves the address of a complex number storage location on the stack.

XCONSTANT (cmplx ---)
Create a named entity that, when

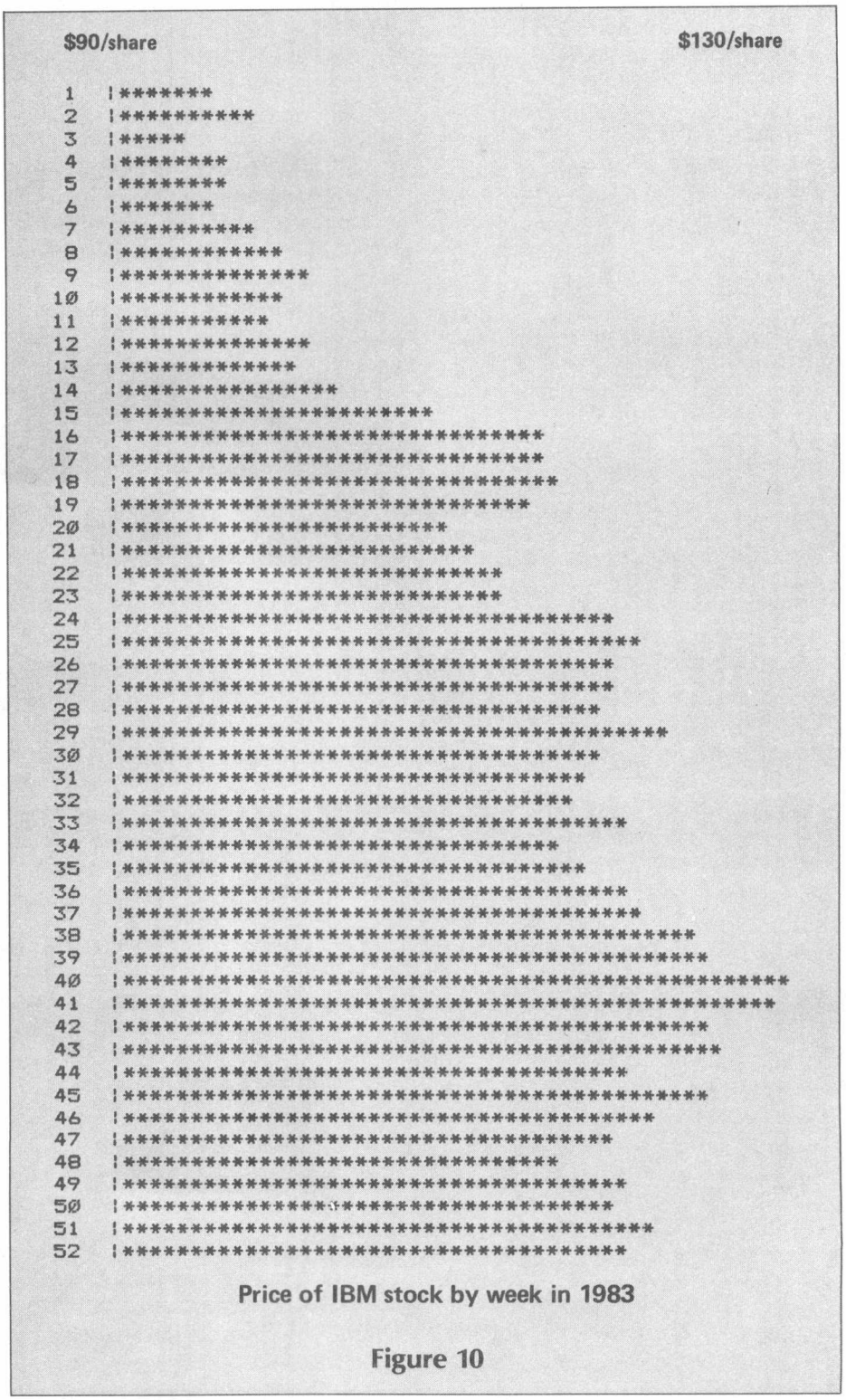


Figure 10

executed, leaves a complex number on the stack.

XDUP (cmplx --- cmplx cmplx)
Duplicate the complex number on the stack.

XSWAP (cmplx2 cmplx1 --- cmplx1 cmplx2)
Swap the two complex numbers on the stack.

XDROP (cmplx ---)
Drop one complex number from the stack.

XOVER (cmplx2 complx1 --- cmplx2 cmplx1 cmplx2)
Copy the complex number under the top of the stack onto the top of the stack.

X2DUP (cmplx2 cmplx1 --- cmplx2 cmplx1 cmplx2 cmplx1)
Duplicate two complex numbers on the stack, keeping their relative positions.

X+ (cmplx2 cmplx1 --- cmplx2+1)
Add two complex numbers and leave the complex result on the stack.

X– (cmplx2 cmplx1 --- cmplx2–1)
Subtract the top complex number from the one beneath and leave the complex result.

X2/ (cmplx --- cmplx/2)
Divide the complex number on the stack by 2 and leave the complex result.

| X |^2 (cmplx --- dbl)
Leave the magnitude squared of the complex number on the stack. The magnitude squared is not a complex number but a double-precision integer.

| X | (complx --- sng)
Leave the single-precision magnitude of the complex number on the stack.

X' (cmplx --- cmplx)
Take the complex conjugate of the complex number on the stack and leave the complex result. (Change the sign of the imaginary part.)

X0= (cmplx --- flag)
Test the complex number on the stack for equivalence with zero and leave the flag on the stack.

X* (cmplx2 cmplx1 scale --- cmplx2*1/scale)
Multiply two complex numbers and then divide the complex result by the single-precision scaling constant, leaving a complex result on the stack.

X*! (addr2 addr1 n ---)
Both addresses point to complex numbers. Multiply the two complex numbers and divide by the single-precision scaling constant. Store the complex result into the area pointed to by addr1.

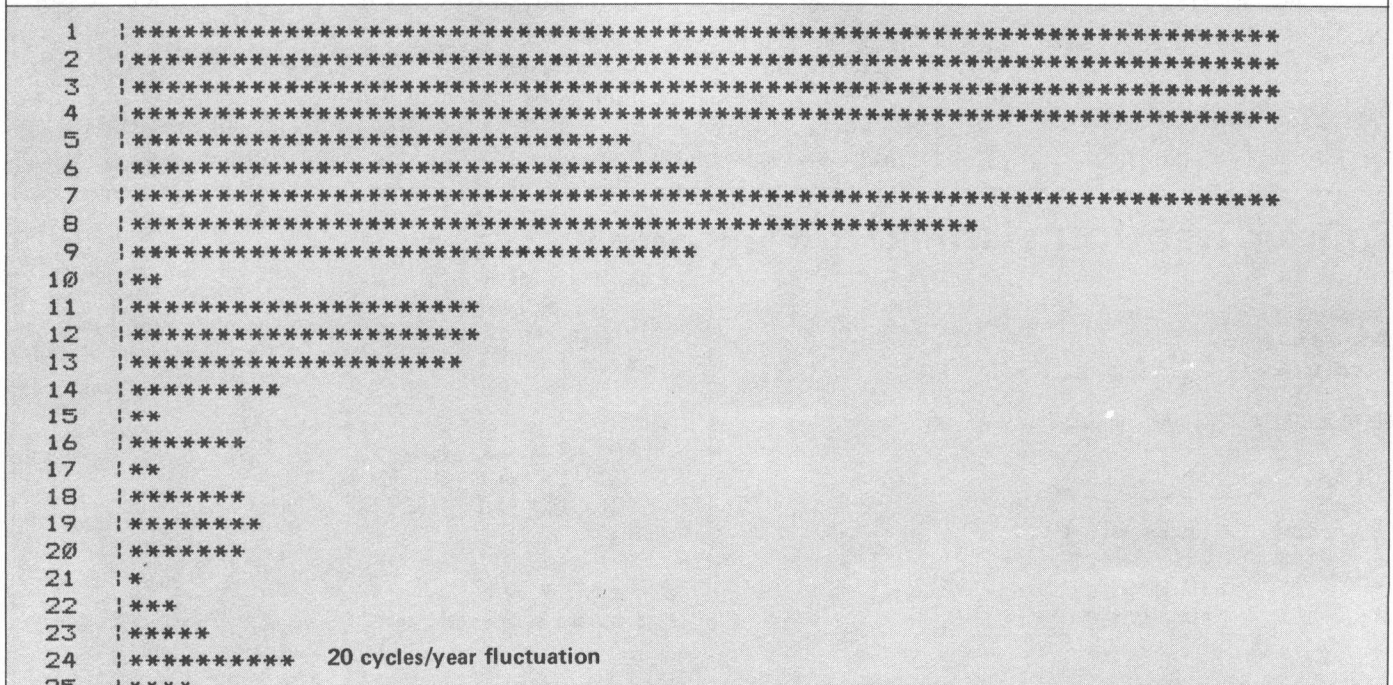
X/ (cmplx2 cmplx1 scale --- cmplx2/1 * scale)
Divide the top complex number into the one beneath and then multiply the complex result by the single-precision scaling constant, leaving a complex result on the stack.

VECTOR (order scale size ---)
Create a named entity that leaves its address when executed. The size of the data area is determined by the number of entries (order) and the size of each (size). The single-precision scaling constant associated with the data is saved in the structure.

&ORDER (vector --- order)
Given the address of a vector, leave the single-precision number of entries on the stack.

&ESIZE (vector --- size)
Given the address of a vector, leave the single-precision size of each element on the stack.

&SCALE (vector --- scale)
Given the address of a vector,



FFT of IBM stock price
Figure 11

leave the single-precision scaling constant of the data on the stack.

&SIZE (vector --- tot_size)
Given the address of a vector, leave the single-precision total number of bytes used by the data area of the vector.

[I] (indx vector --- addr)
Return the address of the element number that corresponds to index. If the requested element is out of bounds, generate an error and halt the program. All vectors are presumed to start with element one.

XVECTOR (order scale ---)
When executed, create a complex vector (see VECTOR) with order elements and set the scaling constant to scale.

X@V (indx vector --- cmplx)
Retrieve the element from the complex vector and leave the complex result on the stack.

X!V (cmplx indx vector ---)
Store the complex element into the complex vector.

10K (--- 10000)
Scaling constant for the trignometric constants in WTAB.

DIR (--- addr)
Variable that contains a flag that represents the direction of the transform—forward (false or 0) or inverse (true or –1).

VEC (--- addr)
Variable that contains the address of the vector undergoing transformation.

N (--- addr)
Variable that contains the size (or order) of the transformation. The same as the number of points in the data array.

U (--- addr)
Complex variable used only be the FFT routine.

W (--- addr)
Another complex variable used only by the FFT routine.

V[I][J] (j i --- addrj addri)
Given two indices, compute the address of the elements in the current vector (held by VEC). Leave the addresses on the stack.

XSWAPV (j i ---)
Swap the two complex numbers in the vector (held by VEC) pointed to by i and j.

BIT-REVERSE (---)
Reorder the complex vector (held by VEC) in preparation for the FFT. For an example of a bit-reversed vector, see the text.

BFLY (j i ---)
Perform a butterfly computation on elements i and j of the vector in VEC. The butterfly is the form:
X[i] = X[j] – X[i]*U
X[j] = X[j] + X[i]*U
If the direction is inverse, then the results are divided by 2.

2** (sgl --- sgl)
Raise 2 to the power of the single-precision number on the stack. Leave the single-precision result. WSIZE lets this definition work on 16- or 32-bit systems.

LOG2 (sgl --- sgl)
Leave the logarithm (base 2) of the single-precision number on the stack. Again WSIZE is for portability.

WTAB (---)
A complex vector that holds the sines and cosines for the angles of the form pi/*N*, where *N* is the number of data points (order of the FFT).

INNER-LOOP (start incr ---)
Perform butterflies on the vector (in VEC) beginning at index "start" and adding "incr" each time. The butterfly is performed between index "I" and "I + incr/2," where I is the loop counter.

FFT-KERNEL (---)
Outer loop of the FFT program. Each loop sets the value of W and U and controls the action of the inner loop.

INIT (vector ---)
Initializes the variable VEC to contain the address of the vector and *N* to contain the number of elements.

DO-VECTOR (vector ---)
Perform a complete FFT or IFFT on the vector on the top of the stack.

FFT (vector ---)
Initialize the direction of the transform (variable DIR) and call DO-VECTOR.

IFFT (vector ---)
Initialize the direction of the transform and call DO-VECTOR.

DDJ

Fourier Transform Listing (Text begins on page 34)

```
Screen # 6
( Load screen )

FORTH DEFINITIONS DECIMAL

: TASK ;

7       LOAD    ( complex math words )
15      LOAD    ( vector words )
19      LOAD    ( fft words )
30      LOAD    ( demo fft words )
```

```
Screen # 7
( square and square root )
( from Forth Dimensions Vol. IV #1, pg 9-10 by Klaxon Suralis )
: D2*       2DUP D+ ;

: EASY-BITS
  0 DO  >R D2* D2* R@ - DUP 0<
        IF   R@ + R> 2* 1-
        ELSE      R> 2* 3 +
  THEN LOOP ;

: 2'S-BIT
  >R D2*  DUP 0<
  IF   D2* R@ -  R> 1+  ELSE D2* R@ 2DUP  U<
       IF   DROP R> 1-  ELSE  - R> 1+ THEN
  THEN ;
-->
```

```
Screen # 8
( square and square root, cont. )

: 1'S-BIT
  >R DUP 0<
  IF   2DROP R> 1+
  ELSE D2*  32768 R@ DU< 0= R> +
  THEN ;
```

```
: SQRT          ( d --- s )
  0 1 8 EASY-BITS  ROT DROP 6 EASY-BITS
  2'S-BIT  1'S-BIT ;

: SQR       DUP M* ;          ( s --- d )
-->
```

Screen # 9

```
( complex data type and operations )

: X@        2@ ;              ( addr  ---  x )
: X!        2! ;              ( x  addr --- )

: XVARIABLE             ( --- )
  CREATE    WSIZE 2* HERE OVER  ( two elements per complex no.)
            ERASE ALLOT         ( initialized to zero )
  DOES> ;

: XCONSTANT             ( x --- )
  CREATE    HERE WSIZE 2* 2DUP      ( two elements per cmplx )
            ERASE ALLOT  X!         ( save TOS into addr )
  DOES>     X@ ;                    ( return complex constant)
-->
```

Screen # 10

```
( complex stack words )

: XDUP      2DUP ;              ( x  ---  x  x )
: XSWAP     2SWAP ;             ( x1  x2  ---  x2  x1 )
: XDROP     2DROP ;             ( x  --- )
: XOVER     2OVER ;             ( x1 x2 --- x1 x2 x1 )
: X2DUP     2OVER 2OVER ;       ( x1  x2  ---  x1  x2  x1  x2 )
-->
```

Screen # 11

```
( complex add, subtract, magnitude )

: X+        ROT + >R + R> ;             ( x1  x2  ---  x1+x2 )
: X-        ROT SWAP - >R - R> ;        ( x1  x2  ---  x1-x2 )
: X2/       2/ SWAP 2/ SWAP ;           ( x --- x/2 )
: |X|^2     SQR ROT SQR D+ ;            ( x --- mag_x**2 )
: |X|       |X|^2 SQRT ;                ( x --- mag_x )
: X'        SWAP NEGATE SWAP ;          ( x --- x_conjugate )
: X0=       0= SWAP 0= AND ;            ( x --- t/f )
-->
```

Screen # 12

```
( complex multiply )
-->

: X*        >R  X2DUP           ( x1  x2  n ---  x1*x2/n )
            ROT R@ */                 ( real part = )
            ROT ROT R@ */  -          ( re1*re2 - im1*im2 )
            R> SWAP >R >R             ( save partial result)
            ROT ROT R@ */             ( re2*im1 )
            ROT ROT R> */  +          ( + re1*im2 )
            R> ;                      ( imag real --- )
```

This word is replaced by one on the screen that follows. The high level FORTH definition here is for documentation purposes.

Screen # 13

```
( assembly def for complex multiply )

  ASM     ( load assembler if not already loaded. )
  CODE X*
      (SP)+ D0 MOVE,  (SP) D1 MOVE,  4 d(SP) D1 MULS,
      2 d(SP) D2 MOVE,  6 d(SP) D2 MULS,  D2 D1 L. SUB,
      D0 D1 DIVS,  (SP) D2 MOVE,  6 d(SP) D2 MULS,
      2 d(SP) D3 MOVE,  4 d(SP) D3 MULS,  D3 D2 L. ADD,
      D0 D2 DIVS,  8 # SP ADDQ,  D2 -(SP) MOVE,  D1 -(SP) MOVE,
      NEXT, END-CODE

  -->
```

Stack indexing is used to keep the original data available as the complex product is built. When the product is complete, the original data is popped off the stack and the product pushed onto it.

Screen # 14

```
( more complex operations )

: X*!       OVER >R >R          ( ^x1 ^x2 n --- x2 <-- x2*x1/n)
            X@  ROT X@  R>              ( form x1*x2/n )
            X*   R> X! ;                ( save it into x2^ )

: X/        >R XSWAP XOVER      ( x1 x2  n  ---  n*x1/x2 )
            X' R@ X*                    ( x1*x2' )
            XSWAP |X| DUP R@ */         ( sqr|x2| )
            SWAP OVER  R@ SWAP */       ( real part )
            ROT ROT R> SWAP */ SWAP ;   ( imag part )
```

Screen # 15

```
( Vector definition )

: VECTOR        ( order scale word_size --- )
  CREATE    ROT 2DUP , ,          ( store order, then wsize )
            ROT ,                 ( then store scale )
            * DUP ,               ( bytes of entry storage )
            HERE OVER ERASE       ( erase vector )
            ALLOT                 ( allot dictionary for entries)
  DOES> ;                         ( return address of base )

-->
```

This data structure holds a vector that contains "order" entries. Legal indicies are in the range of 1 to "order". "Scale" is used in numeric operations on entries, and the "word_size" field describes the number of bytes per entry.

Screen # 16

```
( vector operations )

: &ORDER        ( vector_descriptor --- order_of_vector )
  @ ;

: &ESIZE        ( vec_desc --- size_of_entries )
  WSIZE + @ ;

: &SCALE        ( vec_desc --- scale_of_vec )
  WSIZE 2* + @ ;

: &SIZE         ( vec_desc --- size_of_data_area )
  WSIZE DUP 2* + + @ ;
-->
```

(Continued on page 46)

Fourier Transform Listing (Listing Continued, text begins on page 34)

Screen # 17

```
( matrix operations, cont. )
-->

: [I]          ( index  vec_desc --- addr_of_entry )
  DUP >R &ORDER                ( save vec_desc, get entrysize)
  OVER < OVER 1 < OR ABORT" Vector bounds exceeded... [I]"
  1-  R@ &ESIZE *              ( index * entrysize )
  WSIZE 4 * + R> + ;           ( +offset to entries )
```

This word calculates a storage address given a column number and a matrix descriptor. If the desired address is outside of the alloted vector storage area, an exception is generated. This word is usually followed by "@" or "!". The following screen contains the assembly language equivalent.

Screen # 18

```
( assembly language version of [i] )

ASM
CODE [I]               ( index vec_desc --- addr )
    (SP)+ 0S MOVE,  0 d(BP, 0S) A0 LEA,          ( a0=vec addr)
    (SP)+ D0 MOVE,  1 # D0 CMPI,  1$ BLT,        ( index<1 )
    (A0) D0 CMP,    1$ BGT,                      ( index>max)
    1 # D0 SUBQ,  2 d(A0) D0 MULS,  D0 0S ADD,   ( word size)
    0 # 0S ADDQ,  0S -(SP) MOVE,  NEXT,          ( success )

1$: -1 # -(SP) MOVE, 2$ # 0S MOVE,               ( force abort")
    0 d(BP, 0S) IP LEA,  NEXT,

2$: ] ABORT" Vector bounds exceeded... [I]" [
END-CODE
```

Screen # 19

```
( complex vector defs )

: XVECTOR       ( order scale --- )
  WSIZE 2* VECTOR ;      ( declare a vector with two words/ent.)

: X@V           ( index vec_desc --- x )
  [I] X@ ;               ( fetch vector entry pointed by index)

: X!V           ( x index vec_desc --- )
  [I] X! ;               ( store into vector at position index)
-->
```

The constant WSIZE (on line 3) returns the word size of the Forth system used. For a 16-bit implementation (like LMI Forth) it is 2, for a 32-bit system (like LMI Forth+) it is 4.

Screen # 20

```
( FFT constants and variables )

10000   CONSTANT        10K             ( scaling constant )
        VARIABLE        DIR             ( direction of fft )
        VARIABLE        VEC             ( current vector base)
        VARIABLE        N               ( points in fft )
        XVARIABLE       U               ( cmplx angle counter)
        XVARIABLE       W               ( cmplx angle incr)
-->
```

Screen # 21

```
( bit reversal of vector )

: V[I][J]              ( i  j  --- [i]  [j] )
  VEC @ [I] SWAP               ( calculate first addr )
  VEC @ [I] SWAP ;             ( then second addr )

: XSWAPV               ( i  j  --- )
  V[I][J]                      ( calculate addresses )
  2DUP SWAP >R >R              ( save one copy on ret stack)
  X@ ROT X@  R> X! R> X! ;     ( swap cmplx numbers )
-->
```

Screen # 22

```
( bit reversal, cont. )

: BIT-REVERSE          ( --- )
  N @ 1 SWAP 1 DO              ( for i=1 to size of vector )
    DUP I > IF                 ( compare indicies )
      DUP I XSWAPV             ( swap vector entries )
    THEN
    N @ 2/ SWAP                ( let incr=points/2 )
    BEGIN   2DUP <             ( while incr < indx )
    WHILE   OVER - SWAP 2/ SWAP ( indx=indx-incr, incr=incr/2)
    REPEAT  +                  ( indx=indx+incr )
  LOOP DROP ;                  ( drop indx )
-->
```

Screen # 23

```
( butterfly calculation )

: BFLY          ( i  j  --- )
  V[I][J] 2DUP >R >R           ( calc actual addresses )
  X@  U X@  10K X*             ( form temp product )
  ROT X@  XSWAP X2DUP          ( prepare stack for + and - )
  X- >R >R  X+ R> R>           ( Fi-temp, Fi+temp )
  DIR @                        ( test for fwd/inv transform )
  IF   XSWAP                   ( IFFT, no divide )
  ELSE X2/ XSWAP X2/           ( FFT, divide by points )
  THEN R> X!  R> X!  ;         ( store products )
-->
```

Screen # 24

```
( 2** and log2 )

: 2**           ( n  ---  2**n )
  0 MAX WSIZE 8 * 1- MIN       ( limit input data )
  1 SWAP 0 ?DO 2* LOOP ;       ( DON'T loop if n=0 )

: LOG2          ( n  ---  log2[n] )
  DUP 0= ABORT" Can't take log of zero. "
  0 SWAP                       ( limit one word integer )
  BEGIN   DUP 0>               ( test upper bit )
  WHILE   2* SWAP 1+ SWAP      ( if zero, shift and incr count)
  REPEAT  DROP                 ( done, drop remainder )
  WSIZE 8 * 1- SWAP - ;        ( return log base 2 of input)
-->
```

Screen # 25

```
( table of cosines and sines for 2**0 to 2**7 )

VARIABLE WTAB   8 WTAB !        ( dummy vector entry )
        4 , 10000 , 32 ,        ( entries, wsize, scale, data)
       -10000 ,  00000 ,  00000 , -10000 ,  07071 , -07071 ,
        09239 , -03827 ,  09808 , -01951 ,  09952 , -00980 ,
        09988 , -00491 ,  09997 , -00245 ,
-->
```

This is a table of cosines and sines for angles of pi/n where n is an integral power of two. The first entry is pi/1, cos is -1 and sin is 0. Next is pi/2, cos is 0 and sin is 1. Entries up to pi/128 currently, but it can be extended for larger FFT's.

Screen # 26

```
( inner-loop )

: INNER-LOOP    ( startpt increment --- )
  DUP 2/                        ( increment/2 is bfly offset)
  ROT N @ 1+ SWAP               ( limits are startpt...N )
  DO    I 2DUP +                ( do a butterfly computation )
        BFLY OVER               ( between I and I+offset )
  +LOOP 2DROP ;                 ' increment loop )
-->
```

Screen # 27

```
( fft-kernel )

: FFT-KERNEL            ( --- ) ( N, VEC, must be set )
  N @ LOG2 1+ 1                 ( do for i=0 to log2[N] )
  DO  0 10K U X!                ( init U to 1+j0 )
      I WTAB X@V                ( get 1/-arg[pi/I])
      DIR @ IF X' THEN W X!     ( take conjugate if IFFT )
      I 2** DUP 2/ 1+ 1         ( increment and offset counter)
      DO   I OVER INNER-LOOP    ( figure innermost loop )
           W U 10K X*!          ( set new value for angle )
      LOOP DROP                 ( drop increment value )
  LOOP ;

-->
```

Screen # 28

```
( fft and ifft )

: INIT          ( vector --- )
  DUP VEC !
  &ORDER N ! ;                  ( initialize size variable)

: DO-VECTOR     ( vector --- )
  INIT                          ( initialize size field )
  BIT-REVERSE                   ( bit swap input )
  FFT-KERNEL ;                  ( perform butterflies )

: FFT         0 DIR ! DO-VECTOR ;
: IFFT       -1 DIR ! DO-VECTOR ;
```

Screen # 29

```
( spare screen )
```

Screen # 30

```
( plotting words for complex vector )

VARIABLE MAG      10K MAG !     ( largest value in vector )
VARIABLE MIDDLE    35 MIDDLE !  ( zero position, in char units)
VARIABLE PLOTBUF   68 ALLOT     ( buffer for building image )

: MAP                   ( n --- n )
  MIDDLE @ -
  35 MAG @ */ ;                 ( map into character coord. )
-->
```

Screen # 31

```
( put char into buffer before printing line )

: BUF!                  ( char n --- )
  69 MIN 0 MAX                  ( limit access to buffer )
  PLOTBUF SWAP ROT FILL ;       ( store character into buffer)

: ?BUF!                 ( char n --- )
  DUP 69 <= OVER 0 >= AND       ( check for out-of-bounds )
  IF   PLOTBUF + C!             ( if in bounds, print char )
  ELSE 2DROP
  THEN ;
-->
```

Screen # 32

```
( plot one line )

: PLOTDAT               ( n --- )
  PLOTBUF 70 BLANK              ( blank out plotter buffer )
  ASCII : 0 MAP ?BUF!           ( "zero" line if in range )
  ASCII * SWAP MAP BUF!         ( plot character for data )
  PLOTBUF 70 -TRAILING TYPE ;   ( print the plot buffer )

: PLOTINDX              ( n --- )
  2 SPACES 5 .R                 ( print index number )
  2 SPACES ASCII ¦ EMIT ;       ( and right-hand bar )
-->
```

(Continued on next page)

Fourier Transform Listing (Listing Continued, text begins on page 34)

Screen # 33

```
( plot magnitude for entire vector )

: PLOTMAG               ( vec_desc --- )
  CR  DUP &ORDER 1+ 1           ( do for each entry in vector)
  DO  I PLOTINDX                ( get entry )
      I OVER X@V                ( get entry )
      !X! PLOTDAT CR            ( plot magnitude value )
  LOOP CR DROP ;                ( drop vec. descriptor )

-->
```

Screen # 34

```
( load data from stack )
( TOS is vector descriptor, next is number of points, then )
( scale factor, data points follow )

: VECTOR!
  OVER >R                       ( save number of data points )
  SWAP OVER !                   ( reset size of vector )
  SWAP OVER WSIZE 2* + !        ( reset scale of vector entries)
  1 R> DO                       ( for each vector entry, )
     SWAP OVER  0 ROT
     I  4 ROLL X!V
  -1 +LOOP DROP ;               ( work backwards thru vector )
-->
```

Screen # 35

```
( data for IBM over 1983, by week )

100 64 XVECTOR IBM
9663    9913    9463    9738    9738    9638    9863    10038
10225   10075   9988    10213   10163   10388   11013   11725
11700   11763   11650   11063   11300   11400   11425   12113
12300   12100   12150   12013   12438   12038   11975   11850
12250   11783   11975   12225   12313   12663   12688   13225
13175   12700   12800   12225   12688   12350   12100   11788
12225   12088   12363   12200   0       0       0       0
0       0       0       0       0       0       0       0
100 64 IBM VECTOR!

( This data is the closing price of IBM common stock on Friday
  of each week for 1983.  Eights are converted to decimal frac-
  tions.  The data is "padded" to 64 elements for the FFT. )
```

End Listing

Computing with Streams

by L. L. Odette

How do you work with an infinite list in finite storage? You use lazy evaluation, one of the concepts the author of this theoretical piece explores via Forth words.

The attraction of Forth is that you can do almost anything you want with it. One of the things I like to do is build word sets to explore programming issues. In this article, I'll present Forth words for experiments with data-flow-oriented computation, based on a data structure called a stream.

Much current interest centers on data-flow programs and machine organizations because of the advantages to be gained from concentrating more on the flow of data in a computation than on the sequence of steps.[1,2] The advantages include a nicer mathematical structure for the program, increased program modularity, and a program organization that facilitates use of parallelism. Besides any other advantages, organizing a program in terms of data flow can be a natural way to solve a problem that can be expressed in signal processing terms.

One way to make the data-flow organization of a computation concrete is to implement a data structure called a stream. Think of a stream as a list—nothing more than a sequence of data objects representing the data flow during a computation. The computation is organized as a composition of processes that take streams as input and produce other streams as output.

The mathematical niceties of stream processing arise from the lack of variables and associated concepts of "state." Computations organized about objects with coupled state variables (von Neumann style[3]) are difficult to make modular and are susceptible to bugs due to side effects of steps in the computation. All that happens in a pure stream computation is that one stream becomes some other stream, so you program in a functional programming style. Pure functional programming prohibits anything that changes the values of variables; it thereby deals with side effects by ruling them out altogether. Functional programming has been promoted as an alternative to von Neumann-style programming[3,4].

Stream computations can also be highly modular. This is because most of the processing procedures can be abstracted to a small number of higher-order procedures. For example, there are stream *filters*, that remove objects from the stream if they satisfy a given condition. There are stream *maps* that apply a given function to each object in the stream. There are also stream *accumulate* operations that combine stream objects using some procedure. Programming with streams is largely a matter of designing the proper composition of filters, maps, and accumulations.

Parallelism can be exploited because processes communicate only through the streams. There is no shared memory, and each reference to the stream is identical, so processes that are computationally parallel could be executed by different machines.

My goal is to illustrate how stream processing can be implemented in Forth, to provide a starting point for those who might experiment with these ideas. The implementation introduces other interesting topics, such as delayed evaluation and demand-driven program execution, which are also part of current computing research areas[1].

L.L. Odette, Boston University, College of Engineering, 110 Cummington St., Boston, MA 02215.

Streams in Forth

So, how can streams be implemented in Forth? Well, the straightforward implementation of streams as lists would not allow the full exploitation of the power of stream processing. If implemented properly, stream processing techniques allow us to model communicating sequential processes, implement mutual recursion, and process potentially infinite lists (as long as we process only finite parts of those lists).

If we are going to process an infinite list, the list can't be computed before it is processed, and some form of delayed evaluation is necessary. Delayed evaluation in the Forth context means that the inner interpreter is redirected so that a Forth word executes a little and is then delayed while other Forth words execute. Program execution proceeds as a quasi-parallel interweaving of words—a natural interpretation of mutually recursive definitions.

Demand-driven execution is associated with delayed evaluation, because the delayed part of a word must be explicitly forced to execute. The only way a result is produced during the computation is by constantly forcing execution of delayed words—a mode of execution sometimes called lazy evaluation. Demand-driven execution occurs when instructions are executed only when the result they produce is needed by another, already selected instruction.

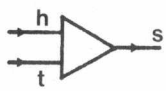
These, then, are the basic ingredients for data-flow processing: a data structure representing the flow of data in a computation, delayed evaluation to suspend computations that produce the elements, and demand-driven execution to force data to be made available when required.

Before discussing the details of the Forth implementation, I'll set the stage with some basic operations and stream processing examples, introducing some useful notation in the process.

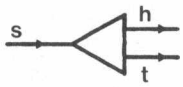
Basic Operations and Notation

Constructors and Selectors

There is a single primitive process that is needed to build streams: the stream constructor. The stream constructor will be represented by a triangle symbol, as follows:



This symbol represents the process of constructing a stream *s* by taking a data object *h* and tacking it onto the beginning of the sequence of data objects *t*. Because streams represent sequences of elements, *h* is the head (first element) of the new sequence *s*, and *t* is the tail (rest) of the sequence. The data flow direction is indicated by arrows; this allows the triangle symbol to do double duty as a representation both of the inverse of the constructor, the stream selector. The representation of the selector is



Note again that the direction of the arrows indicates the direction of data flow. The selector operator takes a stream *s* and makes available the head of the stream *h* and the tail of the stream *t*. In my Forth implementation of stream processes I define the selector words HEAD and TAIL, which expect a stream on the stack, and then leave the head and the tail of the stream, respectively.

It is important to note that while a stream represents a list, the only way to access elements in the list is to build a composition of TAIL operations followed by HEAD. For example, the fourth element of a stream is obtained by

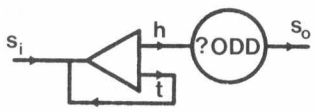
```
TAIL TAIL TAIL HEAD
```

The first TAIL takes a stream and leaves a new stream whose head is the second element of the original stream. The next TAIL leaves a stream whose head is the third element of the original stream. The final TAIL leaves a stream whose head is the fourth element of the original stream and is selected by HEAD.

The selectors and constructors are the only primitive operators on streams. You build higher-order stream processes from them by adding operations on the data elements of the stream. Filters, maps, and accumulations are some of the more common higher-order processes for operating on streams and demonstrate the high level of abstraction possible in stream processing.

Filters, Maps and Accumulations

One common stream process is the *filter*. A filter just takes a stream and returns a new stream by removing from the input stream all the elements satisfying a specified condition. For example, supposing that the word ?ODD tests a number for oddness and returns a flag indicating the result of the test, then the following represents a process that filters out the odd elements of an input stream of numbers.



In the process above, the selector picks out the head of the input stream and ?ODD tests it; then it is passed through to the output if it is even. The tail of the input stream is fed back and the process repeats, selecting the head of the tail, testing for oddness, feeding back the tail of the tail, and so on.

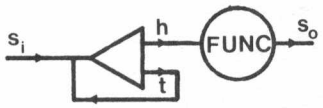
Filters are common elements in data-flow computations; this suggests that an abstraction of the filter above to a generic filter would be useful in designing programs. To build a particular filter from the generic filter, you must combine a predicate (filter word) with the generic filter process. Forth filter definitions would be of the form:

```
: name predicate FILTER ;
```

where *name* is the name given to the filter, and *predicate* represents any word that applies a test to the head of the input stream (leaving true or false on the parameter stack as does the word ?ODD, above). FILTER is the generic filter word, perhaps implemented as an immediate word that sets up a filter at compile time, using the code address compiled for predicate. At runtime the particular filter word expects an input stream on the stack and produces an output stream, filtered as specified by predicate.

A process somewhat similar to a filter is a *map*, a term that takes its name from LISP. What a map does is apply a func-

tion to each element of a stream to produce a new stream. Using the triangle notation, a map process looks like this:



As in the filter process, the selector obtains the head of the input stream s_I and then feeds back the tail. The function FUNC is applied to each element of the input stream to produce the output stream s_O. For example, if the function were one that doubled each element, an input stream 1, 2, 3, 4, . . . would be transformed to the output stream 2 , 4, 6, 8, ... Like filters, maps can be abstracted to an operation combining a function and a generic map. Forth definitions for particular maps would be of the form

: name function MAP ;

where *name* is the name of the MAP process and *function* represents the function word to be applied to each element in the input stream in order to produce the output stream. MAP is the generic map process.

A somewhat more complicated stream process is an *accumulation.* An accumulate process builds an output stream by applying a binary operator to successive elements of the input and output sequences and accumulating the results. Specification of an accumulate process requires the accumulate operator and an initial condition. The head of the output stream is given by the initial condition. The output stream is then fed back to be combined, via the specified operator, with the input stream. The results form the tail of the output stream s_O.

For example, an accumulate based on the + operator would return a stream of partial sums if given a stream of numbers. The partial sum process, based on + with initial condition 0, is shown in Figure 1 (at right). The constructor Element A forms the output stream, starting with 0. Selector Element B picks out elements of the input integer stream in sequence, and Selector C picks out sequential elements of the output stream. Passing the selected elements through + provides the tail for Constructor A by adding elements of the input sequence to corresponding elements of the partial sum sequence.

Filters, maps, and accumulations are some of the basic building blocks for organizing data-flow computations, and using them makes it relatively easy to understand what a stream processing program is supposed to do. There are suggestions that a data-flow analysis of any computation may be couched largely in terms of these operations, as Waters'[5] analysis of the Fortran programs in the IBM Scientific Subroutine Package showed that nearly 60% of the code fit a stream processing description involving filters, maps, and accumulations.

A Prime Example

For a more complicated program involving streams, consider how stream processing operations might be used to construct a list of all prime numbers. One way to generate the primes is to implement a network for the prime sieve of Eratosthenes and then feed it the sequence of integers starting with the integer 2. The output is the sequence of primes.

The sieve of Eratosthenes can be described in data-flow terms as follows:

1) The head of the input stream is prime. Call it *P*.
2) Filter all multiples of *P* from the tail of the input stream.
3) Feed the filtered stream back to (1).

The sequence of *P*'s generated in step 1 is the sequence of prime numbers, starting with the prime number 2. Part 2 of the sieve description is a stream filter that is slightly different from the filter described earlier in that it takes an input number *P* as a parameter and then uses DIVISIBLE-BY-P as the filter predicate. The DIVISIBLE-BY-P filter removes all integers that are divisible by the integer *P* from the input stream. Using a filter of this sort, the stream processor implementing steps 1 through 3 (the sieve) is shown in Figure 2 (below).

The sieve works as follows. In the stream processor diagrammed in Figure 2, the input stream is a sequence of integers 2, 3, 4, ... The selector removes the head of the input stream and sends it to the constructor, so that the head of the input stream becomes the head of the output stream. Thus 2 is the first prime generated by the sieve. This is step 1 in the description above. The selector also sends the head and tail of the input stream to the DIVISIBLE-BY filter. The head of the input stream is the input parameter to DIVISIBLE-BY, so that DIVISIBLE-BY produces the stream 3, 5, 7, 9, 11, 13, 15, . . . — i.e., the tail of the input stream with all multiples of 2 removed. This is step 2 in the description above. In step 3, the output of DIVISIBLE-BY is passed through a prime sieve process to produce the tail of the output stream.

The prime sieve that DIVISIBLE-BY feeds processes the stream 3, 5, 7, 9, 11, 13, 15, . . . as per the sieve description: the head of the input stream becomes the head of the output stream. Thus 3 is the second prime generated by the sieve. DIVISIBLE-BY removes all multiples of 3 from the tail of the input stream to produce the stream 5, 7, 11, 13, 17, . . . — i.e., the stream 2, 3, 4, 5, . . . with all multiples of 2 and 3 removed. The stream 5, 7, 11, 13, 17, . . . is then passed through a sieve to produce the tail of the output stream.

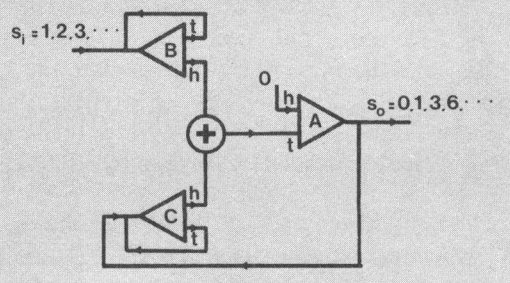


Figure 1

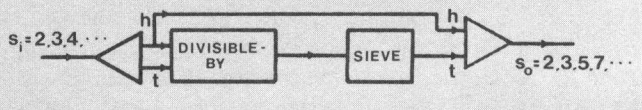


Figure 2

What is unusual about this network is that it contains a copy of itself. The input represents an infinite sequence of integers, as the output stream of primes also represents an infinite sequence, and this means that the above represents a program that is infinitely long. In implementations employing demand-driven execution, only as much of the stream as *is* necessary (demanded as output) is ever computed, and so only as much of the program as is needed exists. During execution, the program in effect builds itself and builds only as much as is required.

The Forth Implementation

The key to a useful implementation of streams and stream processing words is to delay computing anything until the result is needed by some other computation. This is demand-driven computation. To support this mode of execution, the data structure representing a stream will be considered as consisting of a recipe for computing the head and tail of the stream together with the ingredients required by the computation. In Forth, a stream on the stack will consist of *n* stack entries with the structure:

($n-2$ elements), pointer, n . . .

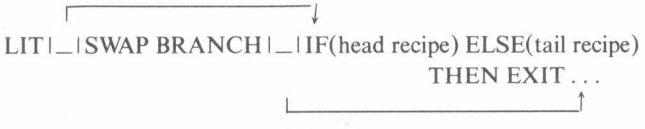
The top stack entry (n) indicates the total number of stack entries comprising the stream. The count is needed in order to manipulate these objects (i.e., we can expect to need words such as STREAM-SWAP, STREAM-DROP, and so on). The next component of the stream is a pointer to the recipe for computing the stream head or tail. The final $n-2$ stack entries comprise the basic ingredients used in the recipe.

The suggested implementation of Forth stream constructor words is given in screen 219 (page 58). The constructors are IMMEDIATE words and can only be used in COLON definitions. They have the form:

```
MAKE-HEAD (head recipe)
MAKE-TAIL (tail recipe)
END STREAM
```

where (head recipe), (tail recipe) represent sets of Forth words that take the ingredients and compute the head and tail, respectively.

At compile time, MAKE-HEAD (head recipe) MAKE-TAIL (tail recipe) END-STREAM is expanded to.



At runtime, MAKE-HEAD expects the parameter stack to contain the stream length and ingredients. The full stream is then constructed by putting a pointer to the IF clause on the stack *and performing* a SWAP. BRANCH delays any computation by transferring control beyond the IF clause. Note that the only effect MAKE-HEAD . . . MAKE-TAIL . . . END-STREAM has at this time is to construct a stream from the data on the stack. Execution of the recipes in the body of the construct is delayed until they are needed to process the stream.

The recipes are forced to execute by the selector words HEAD and TAIL. These are implemented as

```
: HEAD DROP  >R 1 ;
: TAIL DROP  >R 0 ;
```

At runtime they take a stream on the stack, transfer the pointer to the IF (head recipe) ELSE (tail recipe) THEN clause from the parameter stack to the return stack and leave a 1 or 0 on the parameter stack. Control is thereby transferred to the appropriate part of the recipe. When the recipe is complete, control is transferred back to the word following the HEAD or TAIL selector.

A simple example of the use of the stream constructor is provided by the word INTEGERS-FROM, which takes an unsigned integer on the stack and leaves a stream representing all the integers greater than or equal to the input integer. You can then access elements of this (infinite!) list using HEAD and TAIL. INTEGERS-FROM is defined as

```
: INTEGERS-FROM 3 MAKE-HEAD
                  MAKE-TAIL 1 + MYSELF
                  END-STREAM ;
```

where MYSELF is Levan's[6] recursion word.

When executed, INTEGERS-FROM leaves a three-element data structure on the stack—the top element being the count, the next element a pointer, and the third element the starting integer. The recipe for computing the head of the stream is NOP, because the head of the sequence is all that is left on the stack by HEAD. The recipe for computing the tail is to take the head of the sequence, add one, and then form a new stream by executing INTEGERS-FROM.

INTEGERS-FROM represents an infinite sequence in the usual mathematical sense that we never run out of elements of the sequence. The finite word size of stack entries (say, 16 bits) means that in practice INTEGERS-FROM repeats itself every 2^{16} elements, so that, as defined above, it is more accurate to say that INTEGERS-FROM is a representation of all integers greater than or equal to the input integer, MOD 2^{16}.

(MAKE-HEAD(head recipe) MAKE-TAIL(tail recipe) END-STREAM), HEAD and TAIL are the basic stream constructor and selector words. Implementations of stream filters, maps, and accumulations are given in screens 223 – 225 (pages 60 – 61). Other words useful for stream manipulation are given in screens 220 – 222 (pages 58 – 60).

Yet Another Forth Prime Sieve

The sieve of Eratosthenes, implemented with Forth stream processing words, is given in screen 226 (page 61). The basic element of the sieve is DIVISIBLE-BY, a stream filter that removes stream elements that are divisible by some number. DIVISIBLE-BY expects a number and a stream on the stack, filters the stream, and then packages the result in a new stream. The recipe for computing the head of the filtered stream is to drop the top stack entry (the divisor) and execute HEAD. The recipe for computing the tail is

```
>R TAIL R> MYSELF
```

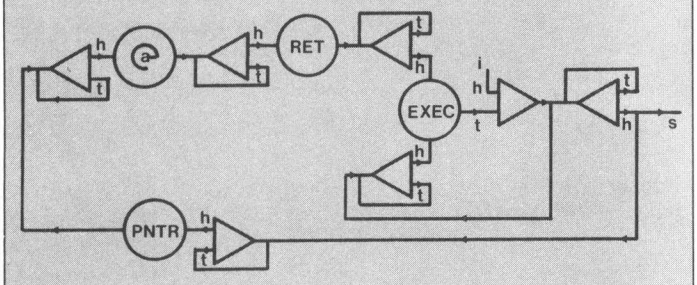
that is, save the divisor, find the tail of the input stream, and filter the tail with DIVISIBLE-BY.

SIEVE processes the input stream according to the following recipes: The head of the output stream is the head of the input stream. The tail of the output stream is found by first filtering the tail of the input stream through DIVISIBLE-BY, using the head of the input stream as the divisor. This filtered stream is then passed through a copy of SIEVE. The recipes follow the three steps in the description of the sieve of Eratosthenes given earlier.

Finding the *n*th prime is straightforward:

```
: PRIMES 2 INTEGERS-FROM SIEVE ;
   (make a stream of all primes)
: SELECT 0 DO TAIL LOOP HEAD . ;
   (select an element of a stream)
PRIMES 10 SELECT.
   (select the 10th prime)
```

There are several noteworthy things about this program. In a sense, the program works back from the request to get the tenth prime, using as much memory as is necessary to compute what has been requested. The larger the prime requested, the greater the memory requirements for the computation.



Each stack is represented by the stack elements and a count. The count is the number of stack elements plus 1, so stack representations can be manipulated by the stream manipulation words STREAM-DUP, STREAM-SWAP, etc. An empty stack is therefore represented by the count, 1. An environment consists of a return stack representation, followed by a parameter stack representation; it is topped with a pointer into the parameter field of the word being interpreted.

The accumulation based on EXEC takes an environment (the output stream) and a code address (input stream) and forms a new environment. The *i* in the figure represents the initial environment for the accumulation.

Figure 3

Data Flow Organization of Forth Inner Interpreter.

Because the stream is on the stack, the available stack space imposes a practical limit on how many primes can be computed via this implementation of the sieve. For example, after 25 primes have been computed, the tail of the output stream is 80 stack elements, representing the input stream 2, 3, 4, . . . with all multiples of the first 25 primes removed. The 80 stack elements, starting from the stack bottom, are three elements specifying the original stream (INTEGERS-FROM), three elements for each invocation of DIVISIBLE-BY (count, recipe pointer, and division) and two elements of the sieve stream header (count, recipe pointer).

Note also the mutual recursion implicit in the fact that DIVISIBLE-BY processes streams that are constructed by SIEVE, while SIEVE processes streams constructed by DIVISIBLE-BY.

A Final Example

Writing programs to experiment with new ways of writing programs could be called metaprogramming. I've presented these Forth words for stream processing as an exercise in metaprogramming, hoping that those who like to experiment in Forth will improve and extend them and that those who solve practical problems with Forth may recognize where similar techniques could be useful in their own programming tasks.

For an application of stream processing on the practical side, I tried designing Forth words to implement a Forth interpreter (screens 227 – 236, page 61 – 67). The programming issues in this task involve deciding how the problem should be posed to fit the data-flow model and choosing a structure for the data. I chose to make the output a stream of environments, each environment consisting of a representation of the parameter and return stacks, plus a pointer into the parameter field of the word being interpreted. The input is an initial value for the environment stream, specifying the contents of data and return stacks and where to start the interpreter. Environment formats are discussed in the legend to Figure 3 (page 56), which illustrates the data-flow structure of the program using triangle notation.

The program is built from three maps and an accumulation. The maps, with their input and output streams, are

MAP	INPUT-STREAM	OUTPUT-STREAM
PNTR	Environments	Parameter field addresses
@	Parameter field addresses	Contents of parameter field addresses(code field addresses)
RET	Code field addresses	Modified code field addresses

The result of applying these three maps in sequence is a three-step process. The first step takes an environment and drops the stack representations, leaving the parameter field pointer (PNTR). Next, the content of that address is obtained (@), yielding a code field address. The stream of code field addresses is then mapped to a modified set of code field addesses (RET) according to the following rule: If the domain word (input to RET) affects the inner interpreter instruction register (LIT, BRANCH, 0BRANCH, etc.) or modifies the return stack (<DO>, <LOOP>, etc.), then the image of the map is the code address of a word that has an effect on the environment data object that is congruent to

the effect of the domain word on the instruction register or return stack. Otherwise, the image code address is identical to the input code address. Screens 229 and 230 contain code for image words of some, but not all, of the words that alter the instruction register or return stack.

The resulting stream of code addresses is fed to an accumulation based on a modified EXECUTE operation.

The result is a program that you can use to single-step through a colon definition, displaying the instruction pointer and the contents of the data and return stacks. Lines 9 – 10 and 13 – 14 of screen 236 show how to prepare a stream to do this. The word STEPS defined in screen 236 expects two environment streams on the parameter stack and will single-step through the two colon definitions in parallel. Extension to $n > 2$ words in parallel is left as an exercise. You may wish to experiment with parallel execution of words that communicate through some shared variable. Other worthwhile exercises include translating the words of screens 220 – 221 to machine code. This will significantly increase speed.

The point of this example is simply that a certain conceptual clarity results from organizing a program so that the data-flow structure of the problem is manifest in the procedures. The description of the operation of the interpreter as I've given it—a sequence of parameter field addresses is generated, the contents of parameter field addresses are used to generate a sequence of code field addresses, and the sequence of code field addresses are executed—has parts that are clearly separable and identifiable in the stream implementation. It follows that the network representation of the data flow given in Figure 3 leads fairly directly to a program. Metaprogramming involves thinking about how direct the transformation is and how useful the result.

Acknowledgements

I would like to thank John Kalafatas of Advanced Micro Devices for his support and for arranging a loan of computer equipment, which greatly facilitated this work.

References

1. Treleaven, P. C., Brownbridge, O. R., and Hopkins R. P., 1982. "Data-driven and demand-driven computer architecture." *ACM Computing Surveys* 14 (1) : 93 – 143.
2. Lerner, E. J., 1984. "Data flow architecture." *IEEE Spectrum* 21 (4) : 57 – 64.
3. Backus, J., 1978. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *Comm. ACM* 21 (8) : 613 – 641.
4. Henderson, P., 1980. *Functional programming.* Prentice-Hall International, Inc., London.
5. Waters, R. C., 1979. "A method for analysing loop programs." *IEEE Trans. on Software Engineering* 5 (3) : 237 – 247.
6. LeVan, J., 1980. "Recursion—the eight queens problem." *Forth Dimensions* 2:6.

DDJ

Streams Listing (Text begins on page 50)

```
SCR #219
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1 \ Stream Constructor and Selector Words
  2
  3 : MAKE-HEAD
  4    HERE 10 + [COMPILE] LITERAL     \ Compile IF clause address
  5    COMPILE SWAP
  6    COMPILE BRANCH HERE 0 , 2       \ Delay is branch around IF
  7    [COMPILE] IF ; IMMEDIATE        \ Rest looks like IF
  8
  9 : MAKE-TAIL   [COMPILE] ELSE ; IMMEDIATE
 10
 11 : END-STREAM  [COMPILE] THEN       \ Terminate If..Else..Then
 12    COMPILE EXIT [COMPILE] THEN ;   \ Terminate delaying branch
 13    IMMEDIATE
 14
 15 :  HEAD DROP >R 1 ;          : TAIL  DROP >R 0 ;


SCR #220
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 : MYSELF   LATEST PFA CFA , ; IMMEDIATE
  3
  4 \  Misc. Stream manipulation words. Each Stream is in the form:
  5 \              (n-2 elements),pointer,n...
```

```
 6 \  where the count (n) is on the top of the parameter stack.
 7 \  *** Translation to machine code is recommended ***
 8
 9 : STREAM-DUP   DUP 1+ DUP 1  DO DUP PICK SWAP LOOP  DROP ;
10
11 : STREAM-OVER  DUP DUP 2+ PICK   SWAP OVER + 1+ SWAP
12                0 DO DUP PICK SWAP LOOP DROP ;
13
14 : STREAM-SWAP  DUP DUP 2+ PICK   SWAP OVER + 1+ SWAP
15                0 DO DUP ROLL SWAP LOOP DROP ;


SCR #221
 0 ( L. L. Odette         6-6-84                          MVP-FORTH )
 1
 2 \ More misc. Stream manipulation words
 3
 4 : STREAM-DROP  DUP   0 DO DROP LOOP ;
 5
 6 : STREAM->R  R> OVER DUP
 7              BEGIN   4 ROLL >R   1- DUP 0= UNTIL
 8              DROP >R >R ;
 9
10 : STREAM-R>  R> R>
11              BEGIN   R> SWAP   1- DUP 0= UNTIL DROP
12              DUP 1+ ROLL >R ;
13
14
15
OK


SCR #222
 0 ( L. L. Odette         6-6-84                          MVP-FORTH )
 1
 2 ( Useful stream manipulations )
 3
 4 : SPLIT   STREAM-DUP TAIL STREAM-SWAP HEAD ;
 5
 6 : DELAY-ADDR   HERE 24 + [COMPILE] LITERAL ; IMMEDIATE
 7
 8 \ Represents stream tail, calculation delayed once
 9 : DELAY-TAIL
10    OVER  DELAY-ADDR -                  \ Don't delay twice
11     IF   DUP 2+   MAKE-HEAD  TAIL HEAD
12                   MAKE-TAIL  TAIL TAIL END-STREAM
13     ELSE   DDROP TAIL MYSELF   THEN ;
14
15 : DELAY-SPLIT   STREAM-DUP DELAY-TAIL STREAM-SWAP HEAD ;

SCR #223
 0 ( L. L. Odette         6-6-84                          MVP-FORTH )
 1
 2 : <FILTER>   ( Stream,Predicate-address . . . Stream )
 3    >R BEGIN                          \ Save predicate and loop
 4        STREAM-DUP HEAD R@ EXECUTE    \ Test head
 5       WHILE TAIL REPEAT              \ Filter tail while true
 6    R>   OVER 3 +                     \ Recover predicate
 7    MAKE-HEAD    DROP HEAD            \ Compute first element
 8    MAKE-TAIL    >R TAIL R> MYSELF    \ Compute Remainder
 9    END-STREAM ;
```

(Continued on next page)

Streams Listing (Listing Continued, text begins on page 58)

```
 10
 11 : FILTER
 12   -2 ALLOT HERE @   [COMPILE] LITERAL    \ Pass predicate address
 13   COMPILE <FILTER> ; IMMEDIATE           \ to generic filter
 14
 15


SCR #224
  0 ( L. L. Odette         6-6-84                        MVP-FORTH )
  1
  2 : <MAP>   ( Stream,Function-address . . . Stream )
  3   OVER 3 +                               \ Format for constructor
  4   MAKE-HEAD  >R HEAD R> EXECUTE          \ Apply function to head
  5   MAKE-TAIL  >R TAIL R> MYSELF           \ Map tail
  6   END-STREAM ;
  7
  8 : MAP
  9   -2 ALLOT HERE @   [COMPILE] LITERAL    \ Pass function address
 10   COMPILE <MAP> ; IMMEDIATE              \ to generic map
 11
 12
 13
 14
 15


SCR #225
  0 ( L. L. Odette         6-6-84                        MVP-FORTH )
  1
  2 \ Length refers to # of items in (init-value,operator,length)
  3
  4 : ACCUMULATE  ( Init-value,operator,length,stream . . . Stream )
  5   DUP 1+ PICK OVER + 2+  \ Compute data length for constructor
  6   MAKE-HEAD              \ (init,op,length) looks like a stream
  7    STREAM-DROP DDROP     \ Compute head
  8   MAKE-TAIL
  9    STREAM-DUP DELAY-TAIL STREAM->R \ Save delayed tail
 10    DUP 2+ DUP ROLL SWAP ROLL SWAP  \ Get operator,length
 11    >R SP@ 3 PICK 2* +  >R >R       \ Save them with stack data
 12    HEAD R@ EXECUTE                 \ Accumulate output and input
 13    R> SP@ R> SWAP - 2/ R> +  \ Package (init-value,op,length)
 14    STREAM-R> MYSELF          \ Remake accumulation
 15   END-STREAM ;

SCR #226
  0 ( L. L. Odette         6-6-84                        MVP-FORTH )
  1
  2 : DIVISIBLE-BY   ( stream,divisor . . . filtered-stream )
  3     >R    BEGIN STREAM-DUP HEAD R@ MOD  \ Divide head by divisor
  4     0=    WHILE  TAIL  REPEAT           \ Repeat while divisible
  5     R> OVER 3 +                         \ Set count for stream
  6    MAKE-HEAD    DROP HEAD               \ head is first not div
  7    MAKE-TAIL    >R TAIL R> MYSELF       \ Filter again for tail
  8    END-STREAM ;
  9
 10 : SIEVE        DUP 2+                   \ Set up length
 11    MAKE-HEAD   HEAD                     \ First element is prime
 12    MAKE-TAIL   SPLIT DIVISIBLE-BY MYSELF \ Filter tail by head
 13    END-STREAM ;
 14
 15
```

```
SCR #227
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 \ Assorted words to manipulate the parameter field pointer
  3 \ and parameter stack representation during EXEC
  4
  5 : GET-POINT  ( . . . x,y,parameter-field-pointer )
  6   R> R> R> R>  4 ROLL >R ;
  7
  8 : PUT-POINT  ( x,y,parameter-field-pointer . . . )
  9   R> SWAP >R SWAP >R SWAP >R >R ;
 10
 11 : RE-STREAM  ( Restore count word to stack representation )
 12   R> R> R> R> SWAP >R SWAP >R SWAP >R ;
 13
 14 : UN-STREAM  ( Stack representation count word to return stack )
 15   R> R> R> 4 ROLL >R >R >R >R ;

SCR #228
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1 \ More assorted pointer and return-stack manipulations
  2
  3 : *LIT  ( . . . @-pointer ) \ Perform like LIT
  4     R> GET-POINT 2+ DUP
  5     >R SWAP >R SWAP >R SWAP >R @ ;
  6
  7 : SET-POINT   ( . . . )     \ Set according to literal field
  8     R> R> *LIT
  9     GET-POINT 2- 4 ROLL +   PUT-POINT   >R >R ;
 10
 11 : +2-POINT                  \ Increment pointer
 12     R> R> GET-POINT 2+ PUT-POINT >R >R ;
 13
 14 : -2-POINT                  \ Decrement pointer
 15     R> R> GET-POINT 2- PUT-POINT >R >R ;

SCR #229
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 \ Replacement words - acting on environments
  3
  4         : *R*   SWAP >R 1- STREAM-SWAP 1+ R> SWAP STREAM-SWAP ;
  5 ( >R ) : *>R  RE-STREAM *R* UN-STREAM ;
  6 ( R> ) : *R>  RE-STREAM STREAM-SWAP *R* STREAM-SWAP UN-STREAM ;
  7 ( R@ ) : *R@  RE-STREAM STREAM-SWAP OVER SWAP 1+ *R*
  8               STREAM-SWAP UN-STREAM ;
  9
 10 ( EXIT ) : *XT   -2-POINT ;
 11 ( LIT )  : *LT   R> *LIT SWAP >R ;
 12
 13 ( 0BRANCH ) : *0BR  0=  IF SET-POINT ELSE +2-POINT THEN ;
 14 ( BRANCH )  : *BR   SET-POINT ;
 15

SCR #230
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 \ More replacement words
```

(Continued on page 66)

Streams Listing (Listing Continued, text begins on page 58)

```
 3
 4 ( <DO> )     : *DO   SWAP RE-STREAM *R* *R* UN-STREAM ;
 5
 6 ( <LOOP> )     : *LOOP  RE-STREAM  STREAM-SWAP UN-STREAM
 7                         1+ OVER OVER - 0=
 8                         IF   DDROP RE-STREAM 2-
 9                              STREAM-SWAP 2+ UN-STREAM +2-POINT
10                         ELSE RE-STREAM STREAM-SWAP UN-STREAM
11                              SET-POINT
12                         THEN ;
13
14
15


SCR #231
 0 ( L. L. Odette          6-6-84                          MVP-FORTH )
 1 \ Using C.E. Eaker's Case construct. Forth Dimensions II(3):37
 2
 3 : RET-STACK  \ Map ret-stack, pntr-control words to equivalent
 4   CASE
 5   [ FIND R>       ] LITERAL OF  [ FIND *R>    ]  LITERAL  ENDOF
 6   [ FIND >R       ] LITERAL OF  [ FIND *>R    ]  LITERAL  ENDOF
 7   [ FIND R@       ] LITERAL OF  [ FIND *R@    ]  LITERAL  ENDOF
 8   [ FIND EXIT     ] LITERAL OF  [ FIND *XT    ]  LITERAL  ENDOF
 9   [ FIND LIT      ] LITERAL OF  [ FIND *LT    ]  LITERAL  ENDOF
10   [ FIND 0BRANCH  ] LITERAL OF  [ FIND *0BR   ]  LITERAL  ENDOF
11   [ FIND BRANCH   ] LITERAL OF  [ FIND *BR    ]  LITERAL  ENDOF
12   [ FIND <DO>     ] LITERAL OF  [ FIND *DO    ]  LITERAL  ENDOF
13   [ FIND <LOOP>   ] LITERAL OF  [ FIND *LOOP  ]  LITERAL  ENDOF
14   DUP ENDCASE  ;
15

SCR #232
 0 ( L. L. Odette          6-6-84                          MVP-FORTH )
 1
 2 : EXEC  \ Accumulate environments with code address stream
 3    SWAP >R SWAP >R SP@ >R     \ Save pointer and stack depth info
 4    EXECUTE
 5    R> SP@ - 2/ R> + R> 2+ ;       \ Re-form environment
 6
 7
 8 : PNTR   \ Extract pointer from an environment
 9    >R STREAM-DROP STREAM-DROP R> ;
10
11 ( Maps )
12 : PNTR-MAP   PNTR       MAP ;
13 : @-MAP      @          MAP ;
14 : RET-MAP    RET-STACK  MAP ;
15

SCR #233
 0 ( L. L. Odette          6-6-84                          MVP-FORTH )
 1
 2 \ ENV-EXEC takes initial environment and leaves accumulation
 3
 4 : ENV-EXEC ( Ret-stack-rep,Param-stack-rep,Pfa . . . Stream )
 5      OVER DUP 3 + PICK + 3 +          \ Compute length
 6      [ FIND EXEC ] LITERAL SWAP       \ Compose for accumulate
 7      STREAM-DUP  DUP 2+               \ Dummy stream
 8       MAKE-HEAD  DDROP  END-STREAM    \ Tail never accessed
 9      PNTR-MAP @-MAP RET-MAP           \ Pass through maps
```

```
 10      ACCUMULATE ;
 11
 12
 13
 14
 15


SCR #234
  0 ( L. L. Odette         6-6-84                          MVP-FORTH )
  1
  2 \ STEP expects an accumulation based on EXEC and leaves a stream
  3 \ HEAD leaves the head of the input accumulation
  4 \ TAIL leaves a new accumulation
  5
  6 : STEP   DUP 2+                        \ Set stream count
  7     MAKE-HEAD   HEAD
  8     MAKE-TAIL   TAIL HEAD              \ Get first of rest
  9       ENV-EXEC  MYSELF                 \ Compose accumulate & step
 10     END-STREAM ;
 11
 12
 13
 14
 15


SCR #235
  0 ( L. L. Odette         6-6-84                          MVP-FORTH )
  1
  2 ( [environment],number . . .  )
  3 : ENV-DISPLAY   ." WORD  " . CR HEX 5 SPACES
  4   ." Pfa " U. 2 SPACES
  5   ." Parameter Stack -> "  1- ?DUP IF 0 DO U. LOOP THEN 5 SPACES
  6   ." Return Stack -> "    1- ?DUP IF 0 DO U. LOOP THEN
  7   CR DECIMAL ;
  8
  9
 10 : TEST0  ( Test word 0 )
 11   ?DUP  IF  0  DO  2*  LOOP  THEN  ;
 12
 13 : TEST1  ( Test word 1 )
 14   BEGIN  1- DUP 0= UNTIL ;
 15

SCR #236
  0 ( L. L. Odette         6-6-84                          MVP-FORTH )
  1
  2 : STEPS  ( N steps for 2 words in parallel )
  3   0 DO
  4   SPLIT  1 ENV-DISPLAY  STREAM-SWAP     \ Execute and display
  5   SPLIT  2 ENV-DISPLAY  STREAM-SWAP     \ Execute and display
  6   LOOP ;
  7
  8 \ Set up stream to execute TEST0
  9 1 ( empty return stack )  2 2 3 ( data stack is: 2 2 )
 10 ' TEST0   ENV-EXEC  STEP
 11
 12 \ Set up stream to execute TEST1
 13 1 ( empty return stack )  2 2 ( data stack is: 2 )
 14 ' TEST1   ENV-EXEC  STEP
 15
```

End Listing

A Forth Native-Code Cross Compiler for the MC68000

by Raymond Buvel

If you have an 8-bit microcomputer system and want to experiment with the Motorola MC68000, this cross compiler may interest you. One of the problems an experimenter faces in using a new microprocessor is how to develop programs for it. While one solution to this problem is to purchase a new computer system and a new set of software for each of the processors that interest you, this is obviously expensive. Another solution is to use an existing computer and its software to develop programs for the new processor. Compilers that use an existing computer to produce code for another machine are called cross compilers.

The cross compiler presented here is a native code compiler for the MC68000. It loads on top of a host Forth development system (which doesn't have to be running on a 68000) and compiles a subset of the Forth language. The compiler itself is written in Forth and I have isolated the necessary system-dependent parts to a few clearly identified words. I expect that the code presented here can be easily ported to most Forth environments.

If you've been wondering how to get into programming on the Motorola 68000 chip, this Forth package could be the tool that gets you started.

The compiler produces two types of compiled definitions: macros and subroutines. The macro definitions are essentially extensions to the compiler itself and do not directly produce an executable program. Subroutine definitions generate the executable code, which can consist of macro, variable, constant, and subroutine references. The executable code is stand-alone and can be placed in ROM. The destination for the executable code is left entirely up to the user and can be changed by altering the definition of a single word in the compiler. The compiler does not contain any built-in words for doing I/O; this is left up to the user.

Raymond Buvel, Box 3071, Moscow, ID 83843.

In this article I am assuming the reader is familiar with the Forth language. Those not familiar with Forth should get a copy of the excellent book *Starting FORTH* by Leo Brodie. I used *Starting FORTH* as a reference when I developed this compiler, so the Forth words implemented here work as described there. I have presented the detailed description of the operators supported by the compiler in assembly language. Although you need a familiarity with the MC68000 assembly language to understand the design of the compiler, it is not necessary to know assembly language to use the compiler.

Memory Layout

Before considering the compiler, you should know how it uses the memory of the MC68000 address space. The compiler uses four separate areas of MC68000 address space. The pointers to these areas all are maintained in MC68000 registers, so the user can allocate any area of memory for the various functions by properly setting up the registers. Thus, you need not recompile the program to change the memory map. These areas are described below.

Code Pool

Subroutine definitions place their output code in the code pool and update the code pool pointer variable in the compiler (M68PCODE in Listing Two, screen 9, on page 90). During execution of the resulting code, the MC68000 program counter is the pointer to this area of memory. Only relative addressing is used, so you can relocate the code pool simply by moving the code and starting the program at the proper place.

Variable Pool

The memory used by variables and arrays is allocated relative to the variable pool pointer (A5 in the MC68000). The compiler word M68ALLOT is used to allocate space and maintain even address alignment. To avoid address faults, the value placed in A5 must be even. With that restriction, you may place the variable pool anywhere in memory by setting the value of A5. With an appropriate supervisor program, you can produce reentrant modules with this compiler by using A5 to assign a separate space for the local variables each time the module is called.

Data Stack

MC68000 register A6 points to the memory used by the data stack. The stack is maintained using the auto decrement addressing mode to store information on the stack and the auto increment addressing mode to remove information from the stack. For further information on the workings of the data stack, see the stack operator section of Listing One (page 76).

Return Stack

The hardware stack is used for the return stack because most of the return stack operations then become automatic. A7 is the pointer to the return stack. Since there is both a supervisor and a user hardware stack pointer, you can use modules generated by this compiler for both interrupt service routines and user programs.

Compiler Description

The Forth subset that I chose to implement is for a particular hardware configuration, but you could expand it if you require different I/O. I assume that you have a computer with a Forth system running on it. In the following discussion, I will refer to this computer as the host. I also assume that you are using the MC68000 as a coprocessor. This assumption greatly reduces the complexity of the subset to be implemented. All functions that interact with the terminal can be left out, and the host can handle all the interaction with the operating system and peripherals.

A simple bidirectional communication channel is all that is required between the host and the MC68000. I have chosen to implement the arithmetic, stack, memory access, and control operators found in most Forth systems. You can write I/O routines for data transfer between the MC68000 and the host using the primitives provided, since the MC68000 uses memory-mapped I/O.

The compiler generates machine code, so there is no need for an inner interpreter. Because the MC68000 provides the capability of writing position-independent code, all of the code produced by this compiler is position independent unless the user explicitly forces it to be otherwise. Because the code is kept separate from the variable and stack space, the output from the compiler can be put into ROM. It is sometimes desirable to use programs generated with this compiler in host environments other than Forth. Therefore, I have provided a simple output scheme that allows you to send the output code to any device supported by the host.

To avoid conflict with the Forth definitions in the host, most of the compiler is in a separate vocabulary (named M68K) that is accessed only through the defining words. All definitions created with the compiler also are placed in this vocabulary to prevent accidental reference to them while using the host development system. I have given some of the words normally used in Forth different names to avoid conflicts with the host definitions; I will discuss these differences later.

The compiler produces two basic types of definitions: macros and subroutines. Macro definitions do not generate any output code, but when referenced they store the code that implements the macro in the definition currently being compiled. Subroutine definitions generate output code when defined and generate a subroutine call when referenced in another subroutine definition. A macro definition may be referenced in another macro or in a subroutine definition, but a subroutine may not be referenced in a macro definition. The macro definition is the basic building block in the compiler, so I will discuss it in detail before considering constant, variable, and array definitions.

Macros

You create a macro in the same way as you do a Forth colon definition, except that you use :M68MAC and ;M68MAC in place of the : and ; of a Forth definition. The body of the definition consists of executable MC68000 machine code or references to macros, variables, constants, and arrays. For examples of macro definitions, see screens 33 – 43 in Listing Two and the examples below.

Defining a macro activates the M68K vocabulary, creates a Forth header in the dictionary of the host, and reserves space for the code length. The host Forth is in execution mode, so any words referenced in the body of a macro definition are executed immediately. Terminating the macro definition stores the length of the code segment and reactivates the Forth vocabulary. Any subsequent reference to the macro copies the code contained within the macro body into the host dictionary at the location HERE, then the dictionary pointer is updated to point to the memory location following the code.

To illustrate this process, Figure 1 (at left) shows the definition of the macro 2*. First :M68MAC is used to start the definition, create the Forth header for 2* (I am being deliberately vague about the form of the header because that depends on the particular implementation of Forth being used), and allocate space for the code length. Next the macro DUP is called; it copies the code for performing a DUP function into the host dictionary at the location HERE and updates the dictionary pointer by two. Then the macro + is called; it copies its code into the dictionary and updates the dictionary pointer by four. Finally ;M68MAC is used to terminate the definition and compute and store the macro length in the two bytes following the header. In this case, the length is six bytes.

Constants and Variables

Single- and double-precision constants

are compiled as macros containing a single MC68000 instruction to push the value of the constant onto the data stack. For example, the word M68CON is used to define a single-precision constant in Figure 2 (at left). The value of the constant is taken from the host stack and stored in the macro as part of a move-immediate instruction (see Listing One). The word M68DCON is the same, except that it involves a double-precision value.

Variables are defined as single-precision constants that push the variable pool relative address onto the stack for use with the fetch and store operations. Note that the variable pool relative address is a 16-bit signed integer, so the variable pool can be no longer than 32K bytes. The word M68VAR defines a single-precision variable, while the word M68DVAR is for double precision.

Arrays are defined as macros that take the index off the stack, compute the variable pool relative address of that element, then leave the result on the stack. The compiler supports arrays whose elements are either byte, single precision, or double precision. The words M68CARY, M68ARY, and M68DARY, respectively, define these data types. Figure 3 (page 70) shows the definition of a byte array containing five elements. The variable pool pointer is shown before and after the definition of the array. Note that the compiler maintains alignment on word boundaries to avoid address exceptions when the code is executed.

:M68MAC 2* DUP + ;M68MAC

2*	Dictionary header for the macro 2*
00 06	Length of macro code segment
3D 16	MC68000 machine code for DUP
30 1E	MC68000 machine code for +
D1 56	

Figure 1
A macro definition and the resulting dictionary entry

5 M68CON #5 (Defines the constant #5)

#5	Dictionary header for the macro #5
00 04	Length of macro code segment
3D 3C	MOVE.W #5, –(A6)
00 05	Pushes the value 5 onto data stack

Figure 2
A constant definition and the resulting dictionary entry

5 M68CARY EXAMP (Defines the byte array EXAMP)

EXAMP	Dictionary header for the macro EXAMP
00 06	Length of macro code segment
30 3C	Code to add the variable pool relative
00 20	address of the array (20 hex) to the
D1 56	index value on the data stack.

Variable pool pointer:
Before = 0020; After = 0026

Figure 3
An array definition, the resulting dictionary entry, and the effect on the variable pool pointer (Note all values are in hex)

Subroutines

Defining a subroutine activates the M68K vocabulary and creates a Forth header in the dictionary of the host. The code pool relative address of the subroutine is then stored as the first entry of the definition. Compilation proceeds in the same way as in a macro definition, except that references to subroutine definitions are also allowed. When the definition is terminated, a return from subroutine instruction is compiled, the code pool pointer M68PCODE is updated, and the code is sent to the output file and deleted from the dictionary. This leaves only the header and the code pool relative address of the subroutine in the dictionary.

Subsequent reference to the subroutine uses the code pool relative address to compute the relative address required in a branch to the subroutine instruction. Note that the branch instructions on the MC68000 restrict your program to 32K bytes because all of the subroutine calls are back branches; forward referencing is not supported in this compiler.

Since the code pool relative address of a subroutine is stored at the start of the definition, you may reference a subroutine recursively. You must exercise care when doing this, however. Subroutine calls do not create local variables, so a subroutine that stores a value in a variable may not operate properly when called recursively. I recommend keeping all variables on the stack in recursive subroutines. When you do this, make sure the data stack is large enough; because there is no check for stack overflow, something

will be clobbered if the data stack space is too small.

Figure 4 (page 72) illustrates the process of subroutine compilation. The word :M68K is used to start the definition and create the Forth header for 4*; this also sets the code pool relative address of 4* (in this case, it is zero). Next the macro 2* is called twice (this macro 2* is the one defined in Figure 1, not the one actually implemented in the compiler, which is more efficient). Each time 2* is called, the code implementing it is sorted in the host dictionary, and the dictionary pointer is updated. Then ;M68K is used to terminate the definition by compiling a subroutine return instruction, adding the length of the subroutine to the code pool pointer, copying the code to the output file, and deleting the code from the dictionary.

Installation

To install the compiler, you need either a FIG Forth or a Forth-79 system. The installation on a Forth-79 system is a little more involved, so I will cover the FIG installation first. Listing Two contains the complete source for the compiler; you must somehow get these 35 screens into your Forth system. (See the section on availability at the end of this article.)

You must customize a few words in the compiler to your system. In screen 12, a word called HIGH-BYTE takes the top entry off the stack and returns the high byte. You must replace this definition with the code to accomplish this task; it may be necessary to use a code definition on your system. The word M68OUT in screen 18 can be designed to send the generated code to whatever output device or file you want (refer to the note in that screen). The word M68OUT currently prints the code on the screen. If you want to use the external reference capability, you will need to modify the definition in screen 20 to send the output where you want it. If you do not want that feature, simply delete screen 20 entirely.

The compiler is loaded in three sections. The basic compiler and error checking routines are loaded from screens 8 – 24. The program control and looping operations with their associated error checking are contained in screens 25 – 33. The macros that implement the operators supported by the compiler are in screens 34 – 43.

For a Forth-79 system, the above applies, but you must do some additional work. I have used the FIG Forth word ENDIF instead of the Forth-79 word THEN, so on a Forth-79 system the definition of ENDIF given in Listing Two, screen 44, should be used. I have also used the FIG word ⟨BUILDS in a ⟨BUILDS . . . DOES⟩ construction; see screen 44 for a discussion of a definition for ⟨BUILDS. With these two definitions, the compiler should run on a Forth-79 system. However, I cannot guarantee that this is the case. Because my system is a combination FIG Forth with Forth-79 extension, my testing may not have picked up all of the problems. If you have difficulty with this, either get in touch with me or send in a letter to the editor.

```
:M68K 4* 2* 2* ;M68K

   4*      Dictionary header for the subroutine 4*
  00 00    Relative address of subroutine

Code pool pointer:
Before = 0000; After = 000E

Code sent to the output file:
      2*        |      2*          |  RTS instruction
----------------|------------------|-----------------
3D 16 30 1E D1 56 | 3D 16 30 1E D1 56 | 4E 75
```

Figure 4
A subroutine definition, the resulting dictionary entry, and the output code

Source	Code size	Time (10 iter.)	Comments
Screen 8	109 bytes	85 sec	Z80 Forth @ 3.5Mhz
Screen 9	220 bytes	8.7 sec	68000 @ 10Mhz with 2 wait states
Screen 11	216 bytes	8.1 sec	68000 @ 10Mhz with 2 wait states
Assembler	74 bytes	2.1 sec	68000 @ 10Mhz with 2 wait states

Figure 5
Results of benchmark tests shown in Listing Three

Using the Compiler

Before explaining how to use the compiler, I want to point out some of the difficulties you will encounter. Although this compiler uses a subset of Forth, you will find that you must modify most Forth programs before the compiler will accept them. Obviously, you must remove and/or replace constructions that are not supported by the compiler, but many programming techniques used in Forth also will not work because the output code is nothing like the indirect threaded code used in most Forth implementations. Therefore, such practices as modifying the values of constants on the fly will not work, nor can you compile things into the dictionary at runtime since there is no dictionary.

The compiler runs as a collection of words in the host Forth system, but I have not rewritten the Forth word NUMBER to be sensitive to the state of the M68K compiler. Therefore, a number contained within a definition will not be compiled into the definition automatically. Instead it goes on the host stack and must be compiled into the definition with the word LITERAL (or DLITERAL in the case of double

precision). Also note that you must use at least one subroutine definition to get the code to the output file. This should all become clear as I go through the example in Listing Three (page 104).

Listing Three, screen 8, shows the Forth implementation of a benchmark program (Jim Gilbreath, *Byte*, September 1981, page 190); I will use this as a reference to show how the Forth code must be modified to compile properly. Screen 9 shows the same program for the M68K compiler. The first thing to notice is the difference in the definitions of the constants. Zero is used several times in the program. To avoid using LITERAL so many times, I defined a constant #0 in the M68K vocabulary and used that wherever a zero was used. (I could have chosen the name 0 for this constant, since it is redefined only when the M68K vocabulary is active, but that would have made the example more confusing.)

The constant SIZE is used in two ways in the program. SIZE is used as a Forth constant when allocating array space and as an M68K constant within the definition of DO-PRIME. It is defined twice, once in the Forth vocabulary using CONSTANT and once in the M68K vocabulary using M68CON. The definition of the variable FLAGS is a little different from screen 8: the M68K compiler follows the convention in *Starting FORTH* and does not provide initialized variables. Note that the constants 1 and 3 used inside DO-PRIME are compiled using LITERAL, as all numbers inside a definition must be if they are not defined as constants. The resulting code for either a constant or literal is identical; only the compilation process is different. I used :M68MAC to define DO-PRIME because I wanted the entire program to be a single subroutine.

Screen 10 contains an initialization routine required to set the pointers used by the code; it also contains a subroutine TEST, which causes the output code to be generated. TEST also causes the benchmark to be iterated 10 times to conform to the requirements of Gilbreath's test. Note that, because the prime count cannot be printed as in the *Forth* version, it is simply dropped from the stack. You might think that you could eliminate the prime count from the program itself, but that would give a false representation of the execution time of the benchmark.

Screen 11 is the same program as screen 9, rewritten to use the array features of the compiler and to remove a couple of the inefficiencies built into the original program. Screen 12, which is identical to screen 10, is here simply to compile screen 11 without using an indirect LOAD. Also shown in Listing Three is an assembly language version of the benchmark program used for timing and code size comparisons. Figure 5 (page 72) shows the results of the benchmark. Note that we pay a fairly heavy penalty for using a stack-oriented langauge. The programming convenience is worth it in most cases, however.

Figure 6 (page 74) is a description of the words that perform the compilation operations. The list is organized functionally rather than alphabetically. The word described is listed to the left, followed by the host stack image; to the right is an example of the proper usage of the word. Listing One contains the assembly language source for the Forth words supported by this compiler. That listing also, includes a short description of the supported words. The reader should refer to Listing One and *Starting FORTH* to clear up any confusion concerning these definitions. Note that Listing One also describes several operators that are not standard Forth. Operators for doing absolute memory references are described in the memory and I/O section. Absolute subroutine calls and jumps are described in the control operations section.

Debugging Your Programs

No easy way exists to debug programs written for this compiler, so I recommend the following development procedure:

(1) *Write and debug the program in*

:M68K (--) **:M68K xxxx**
Creates a header for the subroutine word xxxx in the M58K vocabulary and sets the variables M68ENTRY and M68PFA. Reference to xxxx within a subroutine definition generates a branch to subroutine using the PC relative addressing mode. The word xxxx may only be referenced within a subroutine definition. Any other usage will produce an error message. So long as no side effects occur, the word xxxx may be referenced recursively. Note that the code may be put into ROM because the stack and variable space are kept separate from the code.

;M68K (--)
Terminates the construction of a subroutine definition and sends the code to the output file. The code for the subroutine is deleted from the host dictionary after it is written out, and only the code pool relative address of the subroutine is retained.

:M68MAC (--) **:M68MAC xxxx**
Creates a header for the macro word xxxx in the M68K vocaulary and sets the compiler variable M68PFA. Reference to xxxx within a definition copies the compiled code into the host dictionary. A macro word may be referenced within the definition of another macro word or within the definition of a subroutine word.

;M68MAC (--)
Terminates the construction of a macro type word, encloses the code, and updates the compiler variables.

M68CON (n --) **n M68CON xxxx**
Defines a macro word xxxx that pushes the value n onto the stack when xxxx is executed. The value n must be on the host stack when M68CON is referenced.

M68DCON (d --) **d M68DCON xxxx**
Defines a double-precision constant in the same way as M68CON.

Figure 6

Forth. Your Forth development system provides a good environment for debugging programs that you develop for this compiler. It is very important at this stage to avoid using any operations not supported by the compiler, since you will just have to remove them later. You should avoid using embedded literals in your definitions—they are rather messy to take care of later.

(2) *Translate the program into a form acceptable to the compiler.* If you have avoided using embedded literals in your definitions, the only changes should be replacing the : and ; as appropriate. You can take care of the emedded literals by defining each one as a constant or using the word LITERAL (or DLITERAL).

(3) *Compile the program and load it into your MC68000 computer.* Make sure that registers A5, A6, and A7 are set properly (either by a supervisor program, by hand, or by using the load instructions provided in the compiler).

The program should now operate properly. If not, step (2) is the most likely place to find the errors. I have frequently encountered errors in resolving program control structures, errors that occur because an embedded literal has not been compiled. This type of error is picked up in the compile phase, so incorrect code is not produced.

Final Comments

This compiler could form the basis for a modular programming environment for the MC68000 microprocessor. The modifications necessary would not be very complicated. To make the compiler generate modular code, you would have to add a new word to the basic compiler. This word should reset all of the compiler variables in screen 9 to their original style and generate an appropriate header to permit the operating system to load and execute the module. Another word to terminate a module and check for errors would be required; the error checking code in the existing compiler can be used as a guide. Note that the absolute addressing operators must be used for all global variables. I recommend that global variables be avoided and that all parameters passed from one module to another be passed on the data stack.

I would like to see floating-point arithmetic added to the compiler, but I am unlikely to do it anytime soon. I would also like to have a MC68000 assembler built into the compiler, so machine code does not have to be typed in hex. (The Forth assembler developed by Michael Perry—*Dr. Dobb's Journal*, September 1983—might be adapted for this purpose.) I encourage anyone who is interested to work on these extensions and publish their work in *Dr. Dobb's* for the rest of us to use.

Please note that I have copyrighted this compiler. However, I am releasing it for personal use and nonprofit distribution. If you sell my compiler as an integral part of a commercial software package, I expect a small royalty. You may sell any code produced by the compiler without notifying me or paying royalties. Other than that, feel free to use it as you see fit and give it to whomever you like. You may also want to send it on to your favorite user group. All I ask is that you acknowledge the source.

Availability

By the time you read this, the compiler should be available from the SIG/M User Group, Box 2085, 177 Hadley Ave., Clifton, NJ 07015 (Disks are 8-inch CP/M and sell for less than $10). The compiler is available on the following RCPM system: Thousand Oaks Technical RCP/M Sysop: Trevor Marshall; System 1: (805) 492-5472; System 2: (805) 493-1495.

I would prefer that you get the compiler from one of the above sources, but I will provide an 8-inch CP/M SSSD disk (sorry, no other formats) for $25. [*We'll keep you updated on other sources as we hear of them. — Ed.*]

DDJ

M68ALLOT (n --) **n M68ALLOT**
Allocates n bytes of space in the variable pool by updating the variable pool pointer variable M68PVAR. Note that this word maintains even-byte alignment so that address exceptions will not occur on 16- and 32-bit memory references. Also note that it is an error if the variable pool becomes longer than 32K bytes, but the compiler will not report this as an error—incorrect code would be generated.

M68VAR (--) **M68VAR xxxx**
Defines a single-precision variable by using the pointer M68PVAR as the parameter n for M68CON. The pointer M68PVAR is then updated by two bytes using M68ALLOT. Execution of the word xxxx leaves the variable pool relative address of the variable on the top of the stack.

M68DVAR (--) **M68DVAR xxxx**
Defines a double-precision variable in the same way as M68VAR except that the pointer M68PVAR is updated by four bytes.

M68ARY (n --) **n M68ARY xxxx**
Defines a single-precision array xxxx that is n elements long. The word xxxx is defined as a macro that takes an element number off the top of the stack and leaves the variable pool relative address of that element.

M68CARY (n --) **n M68CARY xxxx**
Defines a byte array in the same manner as M68ARY.

M68DARY (n --) **n M68DARY xxxx**
Defines a double-precision array in the same manner as M68ARY.

EXTERNAL (--) **EXTERNAL xxxx**
Create an external reference xxxx that contains the code pool relative address of the last subroutine word that was defined. The user can customize this word to produce a file containing an external reference list. Currently, this word creates a constant xxxx in the Forth vocabulary of the host.

Figure 6
Description of compiling words

Forth Compiler (Text begins on page 68)

Listing One

```
                ;
                ;  This file contains the assembler language code for all of the
                ;  operations in the M68K compiler.
                ;
                ;  Register usage....
                ;       A7 - Hardware and return stack pointer
                ;       A6 - Data stack pointer
                ;       A5 - Pointer to variable pool
                ;       A4 - Reserved for future use
                ;
                ;       All other registers are free to be used by any word that needs them
                ;       and are to be considered as altered across word boundaries.
                ;
                ;*******************************************************************************
                ;
                ;  Arithmetic operations
                ;
                ;*******************************************************************************
                ;
                ;  +             ( n1 n2 -- sum )
                ;
301E                MOVE.W  (A6)+,D0                ;Get n2
D156                ADD.W   D0,(A6)                 ;n1 + n2
                ;
                ;  -             ( n1 n2 -- dif )        n1-n2
                ;
301E                MOVE.W  (A6)+,D0                ;Get n2
9156                SUB.W   D0,(A6)                 ;n1 - n2
                ;
                ;  *             ( n1 n2 -- prod )
                ;
301E                MOVE.W  (A6)+,D0                ;Get n2
C1D6                MULS    (A6),D0                 ;n2 * n1
3C80                MOVE.W  D0,(A6)
                ;
                ;  /             ( n1 n2 -- quot )       n1/n2
                ;
4C9E 0003           MOVEM.W (A6)+,D0/D1             ;Get operands sign extended
83C0                DIVS    D0,D1                   ;n1/n2
3D01                MOVE.W  D1,-(A6)
                ;
                ;  */            ( n1 n2 n3 -- n-result )  n1*n2/n3
                ;
321E                MOVE.W  (A6)+,D1                ;Get n3
301E                MOVE.W  (A6)+,D0                ;Get n2
C1D6                MULS    (A6),D0                 ;n2*n1 -> D0
81C1                DIVS    D1,D0                   ;n2*n1/n3 -> D0
3C80                MOVE.W  D0,(A6)
                ;
                ;  /MOD          ( u1 u2 -- u-rem u-quot )
                ;
4280                CLR.L   D0
321E                MOVE.W  (A6)+,D1                ;Get u2
301E                MOVE.W  (A6)+,D0                ;Get u1
80C1                DIVU    D1,D0                   ;u1/u2
4840                SWAP    D0                      ;Interchange remainder and quotient
2D00                MOVE.L  D0,-(A6)                ;Return both on stack



                ;
                ;  MOD          ( u1 u2 -- u-rem )
                ;
4280                CLR.L   D0
321E                MOVE.W  (A6)+,D1                ;Get u2
301E                MOVE.W  (A6)+,D0                ;Get u1
80C1                DIVU    D1,D0                   ;u1/u2
4840                SWAP    D0                      ;Interchange remainder and quotient
3D00                MOVE.W  D0,-(A6)                ;Return remainder on stack
```

```
                ;
                ;  */MOD          ( u1 u2 u3 -- u-rem u-result )  u1*u2/u3
                ;
321E                 MOVE.W  (A6)+,D1                ;Get u3
301E                 MOVE.W  (A6)+,D0                ;Get u2
C0DE                 MULU    (A6)+,D0                ;u2*u1 -> D0
80C1                 DIVU    D1,D0                   ;u2*u1/u3 -> D0
4840                 SWAP    D0
2D00                 MOVE.L  D0,-(A6)
                ;
                ;  U*             ( u1 u2 -- ud )
                ;
301E                 MOVE.W  (A6)+,D0
C0DE                 MULU    (A6)+,D0
2D00                 MOVE.L  D0,-(A6)
                ;
                ;  U/MOD          ( ud u1 -- u-rem u-quot )
                ;
321E                 MOVE.W  (A6)+,D1                ;Get u1
201E                 MOVE.L  (A6)+,D0                ;Get ud
80C1                 DIVU    D1,D0                   ;ud/u1
4840                 SWAP    D0
2D00                 MOVE.L  D0,-(A6)
                ;
                ;  1+             ( n -- n+1 )
                ;
5256                 ADDQ.W  #1,(A6)
                ;
                ;  1-             ( n -- n-1 )
                ;
5356                 SUBQ.W  #1,(A6)
                ;
                ;  2+             ( n -- n+2 )
                ;
5456                 ADDQ.W  #2,(A6)
                ;
                ;  2-             ( n -- n-2 )
                ;
5556                 SUBQ.W  #2,(A6)
                ;
                ;  2*             ( n -- n*2 )
                ;
E1D6                 ASL     (A6)
                ;
                ;  2/             ( n -- n/2 )
                ;
E0D6                 ASR     (A6)
                ;

                ;  ABS            ( n -- abs(n) )
                ;
4A56                 TST.W   (A6)                    ;Test for negative
6C02                 BGE.S   ABS                     ;Skip next instruction if not
4456                 NEG.W   (A6)
                ABS
                ;
                ;  DABS           ( d -- abs(d) )
                ;
4A96                 TST.L   (A6)                    ;Test for negative
6C02                 BGE.S   DABS                    ;Skip next instruction if not
4496                 NEG.L   (A6)
                DABS
                ;
                ;  NEGATE         ( n -- -n )
                ;
4456                 NEG.W   (A6)
                ;
                ;  DNEGATE        ( d -- -d )
                ;
4496                 NEG.L   (A6)
                ;
                ;  D+             ( d1 d2 -- d-sum )
                ;
201E                 MOVE.L  (A6)+,D0
D196                 ADD.L   D0,(A6)
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
            ;
            ;  D-              ( d1 d2 -- d-diff )     d1-d2
            ;
201E              MOVE.L  (A6)+,D0
9196              SUB.L   D0,(A6)
            ;
            ;
            ;*****************************************************************************
            ;
            ;  Stack manipulation
            ;
            ;*****************************************************************************
            ;
            ;  DROP            ( n -- )
            ;
548E              ADDQ.L  #2,A6
            ;
            ;  2DROP           ( d -- )
            ;
588E              ADDQ.L  #4,A6
            ;
            ;  SWAP
            ;
2016              MOVE.L  (A6),D0
4840              SWAP    D0
2C80              MOVE.L  D0,(A6)
            ;
            ;  2SWAP           ( d1 d2 -- d2 d1 )
            ;
2016              MOVE.L  (A6),D0
2CAE 0004         MOVE.L  4(A6),(A6)

2D40 0004         MOVE.L  D0,4(A6)
            ;
            ;  DUP             ( n -- n n )
            ;
3D16              MOVE.W  (A6),-(A6)
            ;
            ;  2DUP            ( d -- d d )
            ;
2D16              MOVE.L  (A6),-(A6)
            ;
            ;  OVER            ( n1 n2 -- n1 n2 n1 )
            ;
3D2E 0002         MOVE.W  2(A6),-(A6)
            ;
            ;  2OVER           ( d1 d2 -- d1 d2 d1 )
            ;
2D2E 0004         MOVE.L  4(A6),-(A6)
            ;
            ;  >R              ( n -- )        Store on return stack
            ;
3F1E              MOVE.W  (A6)+,-(A7)
            ;
            ;  R>              ( -- n )        Remove from return stack
            ;
3D1F              MOVE.W  (A7)+,-(A6)
            ;
            ;  I               ( -- n )        Copies top of return stack
            ;
3D17              MOVE.W  (A7),-(A6)
            ;
            ;  I'              ( -- n )        Copies second item on return stack
            ;
```

```
3D2F 0002       MOVE.W  2(A7),-(A6)
                ;
                ;  J            ( -- n )        Copies third item on return stack
                ;
3D2F 0004       MOVE.W  4(A7),-(A6)
                ;
                ;  Push a constant onto the stack       ( -- n )
                ;
3D3C 0000       MOVE.W  #0,-(A6)
                ;
                ;  Push a double constant onto the stack        ( -- d )
                ;
2D3C 0000       MOVE.L  #0,-(A6)
     0000
                ;
                ;*****************************************************************************
                ;
                ;  Memory and I/O operations
                ;
                ;*****************************************************************************
                ;
                ;  Note.. all references to memory are relative to A5 unless otherwise
                ;         specified.
                ;


                ;
                ;  !            ( n adr -- )
                ;               Store in variable
                ;
301E            MOVE.W  (A6)+,D0
3B9E 0000       MOVE.W  (A6)+,0(A5,D0.W)
                ;
                ;  @            ( adr -- n )
                ;               Get from variable
                ;
3016            MOVE.W  (A6),D0
3CB5 0000       MOVE.W  0(A5,D0.W),(A6)
                ;
                ;  C!           ( c adr -- )
                ;               Store in variable
                ;
301E            MOVE.W  (A6)+,D0
321E            MOVE.W  (A6)+,D1
1B81 0000       MOVE.B  D1,0(A5,D0.W)
                ;
                ;  C@           ( adr -- c )
                ;               Get from variable
                ;
3016            MOVE.W  (A6),D0
4241            CLR.W   D1
1235 0000       MOVE.B  0(A5,D0.W),D1
3C81            MOVE.W  D1,(A6)
                ;
                ;  2!           ( d adr -- )
                ;               Store in variable
                ;
301E            MOVE.W  (A6)+,D0
2B9E 0000       MOVE.L  (A6)+,0(A5,D0.W)
                ;
                ;  2@           ( adr -- d )
                ;               Get from variable
                ;
301E            MOVE.W  (A6)+,D0
2D35 0000       MOVE.L  0(A5,D0.W),-(A6)
                ;
                ;  +!           ( n adr -- )
                ;               Add n to the loaction pointed to by adr
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)
Listing One

```
                ;
301E                MOVE.W  (A6)+,D0
321E                MOVE.W  (A6)+,D1
D375 0000           ADD.W   D1,0(A5,D0.W)
                ;
                ;   M68ARY xxxx  ( n -- )        defines an array xxxx n words long
                ;
                ;
                ;   xxxx         ( n -- adr )    returns the address of the n-th element of xxxx
                ;
303C 0000           MOVE.W  #0,D0           ;Array base address
D056                ADD.W   (A6),D0         ;n + address
D156                ADD.W   D0,(A6)         ;2*n + address
                ;
                ;   M68CARY xxxx ( n -- )        defines an array xxxx n bytes long


                ;
                ;
                ;   xxxx         ( n -- adr )    returns the address of the n-th element of xxxx
                ;
303C 0000           MOVE.W  #0,D0           ;Array base address
D156                ADD.W   D0,(A6)         ;n + address
                ;
                ;   M68DARY xxxx ( n -- )        defines an array xxxx n double words long
                ;
                ;
                ;   xxxx         ( n -- adr )    returns the address of the n-th element of xxxx
                ;

303C 0000           MOVE.W  #0,D0           ;Array base address
3216                MOVE.W  (A6),D1         ;n
E541                ASL.W   #2,D1           ;4*n
D041                ADD.W   D1,D0           ;4*n + address
3C80                MOVE.W  D0,(A6)
                ;
                ;   FILL         ( adr n b -- )
                ;                Fills n bytes of memory beginning at the variable pool
                ;                relative address with the value b.
                ;
301E                MOVE.W  (A6)+,D0        ;Get b
321E                MOVE.W  (A6)+,D1        ;Get n
305E                MOVEA.W (A6)+,A0        ;Get variable pool relative address
D1CD                ADDA.L  A5,A0           ;Compute actual address
6002                BRA.S   $02             ;Enter loop at proper point
10C0        $01     MOVE.B  D0,(A0)+        ;Store b and increment address
51C9 FFFC   $02     DBF     D1,$01          ;Repeat n times
                ;
                ;*****************************************************************************
                ;
                ; Note.. the following words reference absoute memory addresses.  They should
                ;        only be used to reference data and I/O devices that are fixed and
                ;        outside the environment of the compiler.  Under NO conditions should
                ;        these operations be used to reference data structures created by the
                ;        compiler.  The compiler data structures are relocatable and there is
                ;        no easy way to find the current location of these data structures.
                ;        The other operators provided above are much more convenient and
                ;        preserve the relocatability.
                ;
                ;
                ;   AW!          ( n short -- )
                ;                Store n at the location specified by the short address.
                ;
305E                MOVEA.W (A6)+,A0        ;Get address
309E                MOVE.W  (A6)+,(A0)      ;Store n
                ;
                ;   AW@          ( short -- n )
                ;                Get n from the location specified by the short address.
```

```
                ;
3056                MOVEA.W (A6),A0         ;Get address
3C90                MOVE.W  (A0),(A6)       ;Get n
                ;
                ;   AL!             ( n long -- )
                ;                   Store n at the location specified by the long address.
                ;
205E                MOVEA.L (A6)+,A0        ;Get address
309E                MOVE.W  (A6)+,(A0)      ;Store n
                ;
                ;   AL@             ( long -- n )
                ;                   Get n from the location specified by the long address.
                ;
205E                MOVEA.L (A6)+,A0        ;Get address
3D10                MOVE.W  (A0),-(A6)      ;Get n
                ;
                ;   CAW!            ( c short -- )
                ;                   Store c at the location specified by the short address.
                ;
305E                MOVEA.W (A6)+,A0        ;Get address
301E                MOVE.W  (A6)+,D0        ;Get c
1080                MOVE.B  D0,(A0)         ;Store c
                ;
                ;   CAW@            ( short -- c )
                ;                   Get c from the location specified by the short address.
                ;
3056                MOVEA.W (A6),A0         ;Get address
4240                CLR.W   D0
1010                MOVE.B  (A0),D0         ;Get c
3C80                MOVE.W  D0,(A6)
                ;
                ;   CAL!            ( c long -- )
                ;                   Store c at the location specified by the long address.
                ;
205E                MOVEA.L (A6)+,A0        ;Get address
301E                MOVE.W  (A6)+,D0        ;Get c
1080                MOVE.B  D0,(A0)         ;Store c
                ;
                ;   CAL@            ( long -- c )
                ;                   Get c from the location specified by the long address.
                ;
205E                MOVEA.L (A6)+,A0        ;Get address
4240                CLR.W   D0
1010                MOVE.B  (A0),D0         ;Get c
3D00                MOVE.W  D0,-(A6)
                ;
                ;   2AW!            ( d short -- )
                ;                   Store d at the location specified by the short address.
                ;
305E                MOVEA.W (A6)+,A0        ;Get address
209E                MOVE.L  (A6)+,(A0)      ;Store d
                ;
                ;   2AW@            ( short -- d )
                ;                   Get d from the location specified by the short address.
                ;
305E                MOVEA.W (A6)+,A0        ;Get address
2D10                MOVE.L  (A0),-(A6)      ;Get d
                ;
                ;   2AL!            ( d long -- )
                ;                   Store d at the location specified by the long address.
                ;
205E                MOVEA.L (A6)+,A0        ;Get address
209E                MOVE.L  (A6)+,(A0)      ;Store d
                ;


                ;   2AL@            ( long -- d )
                ;                   Get d from the location specified by the long address.
                ;
2056                MOVEA.L (A6),A0         ;Get address
2C90                MOVE.L  (A0),(A6)       ;Get d
                ;
                ;   AFILL           ( long_adr n b -- )
                ;                   Fills n bytes of memory beginning at the long absolute
                ;                   address with the value b.
```

(Continued on page 84)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
                ;
301E                    MOVE.W  (A6)+,D0        ;Get b
321E                    MOVE.W  (A6)+,D1        ;Get n
205E                    MOVEA.L (A6)+,A0        ;Get absolute address
6002                    BRA.S   $02             ;Enter loop at proper point
10C0            $01     MOVE.B  D0,(A0)+        ;Store b and increment address
51C9 FFFC       $02     DBF     D1,$01          ;Repeat n times
                ;
                ;
                ;*************************************************************************
                ;
                ;  Comparison operations
                ;
                ;*************************************************************************
                ;
                ;  MIN          ( n1 n2 -- n-min )
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
3216                    MOVE.W  (A6),D1                 ;n1
B041                    CMP.W   D1,D0                   ;n2-n1
6F02                    BLE.S   MIN
C141                    EXG     D0,D1                   ;Swap if D1 < D0
3C80            MIN     MOVE.W  D0,(A6)
                ;
                ;  MAX          ( n1 n2 -- n-max )
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
3216                    MOVE.W  (A6),D1                 ;n1
B041                    CMP.W   D1,D0                   ;n2-n1
6C02                    BGE.S   MAX
C141                    EXG     D0,D1                   ;Swap if D1 > D0
3C80            MAX     MOVE.W  D0,(A6)
                ;
                ;  =            ( n1 n2 -- f )          if n1 = n2 then f is true
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
321E                    MOVE.W  (A6)+,D1                ;n1
B240                    CMP.W   D0,D1                   ;n1-n2
57C0                    SEQ     D0
0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
                ;
                ;  <            ( n1 n2 -- f )          if n1 < n2 then f is true
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
321E                    MOVE.W  (A6)+,D1                ;n1
B240                    CMP.W   D0,D1                   ;n1-n2
5DC0                    SLT     D0

0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
                ;
                ;  >            ( n1 n2 -- f )          if n1 > n2 then f is true
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
321E                    MOVE.W  (A6)+,D1                ;n1
B240                    CMP.W   D0,D1                   ;n1-n2
5EC0                    SGT     D0
0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
                ;
                ;  D=           ( d1 d2 -- f )          if d1 = d2 then f is true
                ;
201E                    MOVE.L  (A6)+,D0                ;d2
221E                    MOVE.L  (A6)+,D1                ;d1
B280                    CMP.L   D0,D1                   ;d1-d2
57C0                    SEQ     D0
0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
```

```
            ;
            ;  D<           ( d1 d2 -- f )          if d1 < d2 then f is true
            ;
201E             MOVE.L  (A6)+,D0              ;d2
221E             MOVE.L  (A6)+,D1              ;d1
B280             CMP.L   D0,D1                 ;d1-d2
5DC0             SLT     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  D>           ( d1 d2 -- f )          if d1 > d2 then f is true
            ;
201E             MOVE.L  (A6)+,D0              ;d2
221E             MOVE.L  (A6)+,D1              ;d1
B280             CMP.L   D0,D1                 ;d1-d2
5EC0             SGT     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  0=           ( n -- f )              if n = 0 then f is true
            ;  Alternate name is NOT
            ;
4A5E             TST.W   (A6)+
57C0             SEQ     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  0<           ( n -- f )              if n < 0 then f is true
            ;
4A5E             TST.W   (A6)+
5DC0             SLT     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  0>           ( n -- f )              if n > 0 then f is true
            ;
4A5E             TST.W   (A6)+

5EC0             SGT     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  D0=          ( d -- f )              if d = 0 then f is true
            ;
4A9E             TST.L   (A6)+
57C0             SEQ     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  D0<          ( d -- f )              if d < 0 then f is true
            ;
4A9E             TST.L   (A6)+
5DC0             SLT     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  D0>          ( d -- f )              if d > 0 then f is true
            ;
4A9E             TST.L   (A6)+
5EC0             SGT     D0
0240 0001        ANDI.W  #1,D0
3D00             MOVE.W  D0,-(A6)
            ;
            ;  AND          ( u1 u2 -- and )
            ;
301E             MOVE.W  (A6)+,D0
C156             AND.W   D0,(A6)
            ;
            ;  OR           ( u1 u2 -- or )
            ;
301E             MOVE.W  (A6)+,D0
8156             OR.W    D0,(A6)
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
                ;
                ;  XOR          ( u1 u2 -- xor )        Exclusive OR
                ;
301E                MOVE.W   (A6)+,D0
B156                EOR.W    D0,(A6)
                ;
                ;  1'S          ( u -- compl )          One's compliment
                ;
4656                NOT.W    (A6)
                ;
                ;*****************************************************************************
                ;
                ;  Control operations
                ;
                ;*****************************************************************************
                ;
                ;  Note.. all control structures use the PC relative addressing mode
                ;         this makes the code position independent.
                ;
                ;
                ;  IF           ( f -- )
                ;               Takes an entry off the stack and branches if the value

                ;               is FALSE (0).
                ;
4A5E                TST.W    (A6)+                  ;Remove and test flag
6700 ****           BEQ      ELSE
                ;
                ;  ELSE         ( -- )
                ;               Branches around the else part and provides a target for the
                ;               false branch of the if part.
                ;
6000 ****           BRA      ENDIF
                ELSE
                ;
                ;  ENDIF        ( -- )
                ;               Provides a target for the branch around the else part, and if
                ;               the else is missing provides a target for the false branch of
                ;               the if part.
                ;
                ENDIF
                ;
                ;  BEGIN        ( -- )
                ;               Provides a target for a branch back from UNTIL, AGAIN, REPEAT.
                ;
                BEGIN
                ;
                ;  UNTIL        ( f -- )
                ;               Takes an entry off the stack and branches to BEGIN if the
                ;               value is FALSE (0).
                ;
4A5E                TST.W    (A6)+                  ;Remove and test flag
6700 ****           BEQ      BEGIN
                ;
                ;  AGAIN        ( -- )
                ;               Always branches to BEGIN
                ;
6000 ****           BRA      BEGIN
                ;
                ;  WHILE        ( f -- )
                ;               Takes an entry off the stack and branches to REPEAT if the
                ;               value is FALSE (0).
                ;
4A5E                TST.W    (A6)+                  ;Remove and test flag
6700 ****           BEQ      REPEAT
```

```
                ;
                ;  REPEAT       ( -- )
                ;               Always jumps back to begin and provides a target for WHILE.
                ;
6000 ****           BRA     BEGIN
                REPEAT
                ;
                ;  DO           ( limit index -- )
                ;               Remove the index and limit from the data stack and put on
                ;               the return stack.  Provides a target for LOOP and +LOOP
                ;
2F1E                MOVE.L  (A6)+,-(A7)
                DO
                ;
                ;  LOOP         ( -- )

                ;               Increment the index and test for end of the loop.
                ;
5257                ADDQ.W  #1,(A7)                 ;Increment index
4C97 0003           MOVEM.W (A7),D0/D1              ;Get index (D0) and limit (D1)
B041                CMP.W   D1,D0
6D00 ****           BLT     DO                      ;Continue if (index - limit) < 0
588F                ADDQ.L  #4,A7                   ;Drop index and limit
                ;
                ;  +LOOP        ( n -- )
                ;               Add n to the index then
                ;                  IF n > 0 then continue if (index - limit) < 0
                ;                  ELSE continue if (index - limit) >= 0.
                ;
301E                MOVE.W  (A6)+,D0                ;Get increment
D157                ADD.W   D0,(A7)                 ;Update index
4C97 0006           MOVEM.W (A7),D1/D2              ;Get index (D1) and limit (D2)
4A40                TST.W   D0                      ;Test for negative
6E04                BGT.S   $01
B441                CMP.W   D1,D2                   ;Test (limit - index)
6002                BRA.S   $02
B242            $01 CMP.W   D2,D1                   ;Test (index - limit)
6D00 ****       $02 BLT     DO                      ;Continue if (condition tested) < 0
588F                ADDQ.L  #4,A7                   ;Drop index and limit
                ;
                ;  LEAVE        ( -- )
                ;               Terminate loop by seting limit equal to index
                ;
3F57 0002           MOVE.W  (A7),2(A7)
                ;
                ;  Routines for accessing external programs and subroutines.
                ;
                ;
                ;  JSR.W        ( short address -- )
                ;               Jump to subroutine using short address from tos
                ;
305E                MOVEA.W (A6)+,A0
4E90                JSR     (A0)
                ;
                ;  JSR.L        ( long address -- )
                ;               Jump to subroutine using long address from tos
                ;
205E                MOVEA.L (A6)+,A0
4E90                JSR     (A0)
                ;
                ;  JMP.W        ( short address -- )
                ;               Jump to location pointed to by short address on stack
                ;
305E                MOVEA.W (A6)+,A0
4ED0                JMP     (A0)
```

(Continued on page 90)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
                ;
                ;  JMP.L        ( long address -- )
                ;               Jump to location pointed to by long address on stack
                ;
205E                MOVEA.L (A6)+,A0
4ED0                JMP     (A0)
                ;
                ;

                ;*****************************************************************************
                ;
                ;  Initialization operations
                ;
                ;*****************************************************************************
                ;
                ;  Note.. the following words initialize the registers of the M68000
                ;
                ;
                ;  A5LD         Load the variable pool pointer
                ;
2A7C 0000           MOVEA.L #0,A5
     0000
                ;
                ;  A6LD         Load the data stack pointer
                ;
2C7C 0000           MOVEA.L #0,A6
     0000
                ;
                ;  A7LD         Load the return stack pointer
                ;
2E7C 0000           MOVEA.L #0,A7
     0000
                ;
                ;
                    END
```

End Listing One

Listing Two

```
Screen # 0

 0 ( M68K Cross compiler -- Copyright Notice )
 1 ;S
 2 FORTH based cross compiler for the Motorola 68000 microprocessor
 3
 4 Copyright 1983 by   Raymond L. Buvel
 5                     Box 3071
 6                     Moscow, ID 83843
 7
 8 All rights reserved except as stated below.
 9
10 This compiler may be distributed to anyone provided this
11 copyright notice is included and the distribution is not for
12 profit.  Contact me concerning royalties for commercial
13 distribution.  There is no royalty on code produced with this
14 compiler provided the compiler itself is not SOLD as an integral
15 part of a software package.

Screen # 8

 0 ( M68K Cross Compiler -- Vocabulary definition )
 1 VOCABULARY M68K IMMEDIATE
 2 M68K DEFINITIONS
```

```
 3 HEX
 4 -->
 5 Note.. the compilation words listed below are contained in
 6 the FORTH vocabulary and cause entries to be made in the
 7 M68K vocabulary.
 8
 9 :M68K    :M68MAC M68VAR  M68DVAR M68CON  M68DCON
10 M68ARY   M68DARY M68CARY
11
12
13
14
15
```

Screen # 9

```
 0 ( M68K Cross Compiler -- Variable definitions )
 1 M68K DEFINITIONS
 2 ( Code pointer in M68000 -- note relative addressing ! )
 3 0 VARIABLE M68PCODE
 4 ( Variable pool pointer in M68000 -- relative to A5 )
 5 0 VARIABLE M68PVAR
 6 ( Entry point of the subroutine being defined )
 7 0 VARIABLE M68ENTRY
 8 ( Parameter field address [ in HOST ] of word being defined )
 9 0 VARIABLE M68PFA
10 -->
11
12
13
14
15
```

Screen # 10

```
 0 ( M68K Cross Compiler -- Variable definitions )
 1 M68K DEFINITIONS
 2 ( Error checking variables )
 3 0 VARIABLE M68?MAC ( True if in a MACRO definition )
 4 0 VARIABLE M68K?   ( True if in a SUBROUTINE definition )
 5 0 VARIABLE M68?PAIRS ( Count of incomplete branching ops. )
 6 -->
 7
 8
 9
10
11
12
13
14
15
```

Screen # 11

```
 0 ( M68K Cross Compiler -- Error checking )
 1 M68K DEFINITIONS
 2 : ?M68PAIRS ( Check for unbalanced control structures )
 3         M68?PAIRS @ IF
 4           ." Error! unbalanced control structure "
 5           0 M68?PAIRS ! ABORT ENDIF ;
 6 : ?M68K ( Check for errors in compiling a subroutine )
 7         M68K? @ 0= IF ( Check if compiling a subroutine )
 8           ." Error! not compiling a SUBROUTINE "
 9           ABORT ENDIF ;
10 : ?M68MAC ( Check for errors in compiling a macro )
11         M68?MAC @ 0= IF ( Check if compiling a macro )
12           ." Error! not compiling a MACRO "
13           ABORT ENDIF ;
14 -->
15
```

Screen # 12

```
 0 ( M68K Cross Compiler -- Compile constants )
 1 M68K DEFINITIONS
 2 ( n -- c )
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
 3 : HIGH-BYTE 8 SHIFT ;  ( Leave high byte of n on stack )
 4 ( n -- )
 5 : $CON  DUP HIGH-BYTE C, C, ; ( Compile const high-byte first )
 6 ( d -- )
 7 : $DCON $CON ( Compile high word )
 8         $CON ; ( Compile low word )
 9 -->
10 Note.. to transport the compiler to other FORTH systems the
11 word HIGH-BYTE must be written so that it takes the number off
12 the top of the stack and leaves the high byte of that number.
13 On some FORTH systems HIGH-BYTE may have to be a CODE
14 definition.
15


Screen # 13

 0 ( M68K Cross Compiler -- Compiling Words )
 1 M68K DEFINITIONS HEX
 2 ( address -- )
 3 : M68MAC ( Compile MACRO code into any definition )
 4         DUP @ SWAP 2+ OVER HERE SWAP CMOVE ALLOT ;
 5 : M68SUB ( Compile SUBROUTINE code into subroutine definition )
 6         ?M68K 61 C, 00 C,  ( BSR  addr )
 7         HERE M68PFA @ 2+ -  ( Compute code length )
 8         M68ENTRY @ + SWAP @ SWAP - ( Compute displacement )
 9         $CON ; ( Compile displacement )
10 -->
11 Note.. the memory image of a MACRO to be compiled is:
12   addr    Number of bytes of code to compile
13   addr+2  Bytes of code to be compiled.
14 The memory image of a SUBROUTINE to be compiled is:
15   addr    Address of subroutine relative to start of code

Screen # 14

 0 ( M68K Cross Compiler -- MACRO Compiling Words  )
 1 FORTH DEFINITIONS
 2 ( Create header and set compiler variables )
 3 : :M68MAC ( Begin a MACRO definition )
 4         [COMPILE] M68K DEFINITIONS
 5         M68K 1 M68?MAC ! <BUILDS HERE M68PFA !
 6         0 , ( Initialize the number of bytes field )
 7         DOES> M68MAC ;
 8 M68K DEFINITIONS
 9 : ;M68MAC ( terminate a MACRO type definition )
10         ?M68PAIRS ?M68MAC 0 M68?MAC ! ( Error check & reset )
11         HERE M68PFA @ 2+ -  ( Compute code length )
12         M68PFA @ !  ( Store in length field )
13         [COMPILE] FORTH DEFINITIONS ;
14 -->
15

Screen # 15

 0 ( M68K Cross Compiler -- Compiling words - constants    )
 1 FORTH DEFINITIONS HEX
 2 : M68CON ( Define a single precision constant )
 3         :M68MAC 3D C, 3C C, ( MOVE.W #const,-[A6] )
 4         M68K $CON ( Compile constant )
 5         ;M68MAC ;
 6 FORTH DEFINITIONS
 7 : M68DCON ( Define a double precision constant )
 8         :M68MAC 2D C, 3C C, ( MOVE.L #const,-[A6] )
 9         M68K $DCON ( Compile double constant )
10         ;M68MAC ;
11 -->
12
```

```
13
14
15

Screen # 16

 0 ( M68K Cross Compiler -- Compiling words - variables     )
 1 FORTH DEFINITIONS
 2 ( n -- )
 3 : M68ALLOT ( Allot n-bytes in variable pool )
 4         DUP 1 AND IF 1+ ENDIF ( Byte allign )
 5         M68K M68PVAR +! ; ( Update pointer )
 6 FORTH DEFINITIONS
 7 : M68VAR ( Define a single precision variable )
 8         M68K M68PVAR @ 2 M68ALLOT ( Get and update pointer )
 9         M68CON ( Define the address as a constant ) ;
10 FORTH DEFINITIONS
11 : M68DVAR ( Define a double precision variable )
12         M68K M68PVAR @ 4 M68ALLOT ( Get and update pointer )
13         M68CON ( Define the address as a constant ) ;
14 -->
15

Screen # 17

 0 ( M68K Cross Compiler -- SUBROUTINE Compiling Words  )
 1 FORTH DEFINITIONS
 2 ( Create header and set compiler variables )
 3 : :M68K ( Begin a SUBROUTINE definition )
 4         [COMPILE] M68K DEFINITIONS
 5         M68K 1 M68K? ! ( Set to compiling )
 6         <BUILDS HERE M68PFA ! M68PCODE @ DUP
 7         M68ENTRY ! , ( Store subroutine address )
 8         DOES> M68SUB ;
 9 -->
10 Note.. a SUBROUTINE definition may call itself if there are
11 no side effects.  This means that all data altered by the
12 defined word should be on the stack, not stored in variables.
13
14
15

Screen # 18

 0 ( M68K Cross Compiler -- Code output )
 1 M68K DEFINITIONS
 2 ( byte to be sent to code output file -- )
 3 : M68OUT ( Link to the code output file )
 4         BASE @ >R HEX . CR R> BASE ! ;
 5 -->
 6 Note.. the code in the above definition should be replaced
 7 with the appropriate words to send the compiler output to the
 8 code file of your choice.  This could be a disk file, a tape,
 9 your MC68000 computer, a printer, or any other output sink you
10 may want to use.  The protocall is determined by your output
11 word.  The compiler does not assume any protocall so it is a
12 general purpose tool for generating MC68000 code.
13
14
15


Screen # 19

 0 ( M68K Cross Compiler -- SUBROUTINE Compiling Words  )
 1 M68K DEFINITIONS HEX
 2 : ;M68K ( Terminate a SUBROUTINE definition )
 3         ?M68PAIRS ?M68K 0 M68K? ! ( Error check & reset )
 4         4E C, 75 C,  ( Compile an RTS instruction )
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
 5         HERE M68PFA @ 2+ -  ( Compute code length )
 6         DUP M68PCODE +! ( Update code pointer )
 7         M68PFA @ 2+ ( Start of compiled code )
 8         SWAP 0 DO
 9                 DUP C@ M68OUT 1+ ( Output a byte of code )
10         LOOP DROP
11         M68PFA @ 2+ DP ! ( Delete code from dictionary )
12         [COMPILE] FORTH DEFINITIONS ;
13 -->
14
15

Screen # 20

 0 ( M68K Cross Compiler -- EXTERNAL )
 1 M68K DEFINITIONS
 2 : $EXTERNAL ( Define entry point as a constant in FORTH voc.
 3         [COMPILE] FORTH DEFINITIONS
 4         M68ENTRY @ CONSTANT ;
 5 FORTH DEFINITIONS
 6 : EXTERNAL ( Compile an external reference )
 7         M68K M68K? @ M68?MAC @ OR
 8         IF ." Can't use EXTERNAL while compiling"
 9            CR ABORT ENDIF
10         $EXTERNAL ;
11 -->
12 Note.. to send the external reference list somewhere else,
13 replace $EXTERNAL with the appropriate word.  Make sure its
14 function is equivalent to the above, i.e. it must take the
15 next word in the input stream as the identifier.

Screen # 21

 0 ( M68K Cross Compiler -- Words - literals )
 1 M68K DEFINITIONS HEX
 2 : LITERAL ( Define a single precision literal )
 3         3D C, 3C C, ( MOVE.W #const,-[A6] )
 4         $CON ; ( Compile constant )
 5 : DLITERAL ( Define a double precision literal )
 6         2D C, 3C C, ( MOVE.L #const,-[A6] )
 7         $DCON ; ( Compile double constant )
 8 : BYTES 0 DO 20 WORD HERE NUMBER DROP C, LOOP ;
 9 -->
10 Note.. Used as n BYTES followed by bytes to be compiled into
11 the HOST dictionary.  This word may be used within a :M68K
12 or :M68MAC definition but NOT within a colon definition.
13
14
15


Screen # 22

 0 ( M68K Cross Compiler -- Compiling words - arrays )
 1 M68K DEFINITIONS HEX
 2 ( adr -- )
 3 : $M68ARY ( Define code for a single precision array )
 4         :M68MAC 30 C, 3C C, ( MOVE.W #const,D0 )
 5         $CON ( Compile address )
 6         D0 C, 56 C, D1 C, 56 C,
 7         ;M68MAC ;
 8 ( adr -- )
 9 : $M68DARY ( Define code for a double precision array )
10         :M68MAC 30 C, 3C C, ( MOVE.W #const,D0 )
11         $CON ( Compile address )
12         32 C, 16 C, E5 C, 41 C, D0 C, 41 C, 3C C, 80 C,
13         ;M68MAC ;
14 -->
15
```

```
Screen # 23

 0 ( M68K Cross Compiler -- Compiling words - arrays )
 1 FORTH DEFINITIONS
 2 ( n -- )
 3 : M68ARY ( Define a single precision array n cells long )
 4         M68K M68PVAR @ ( Get base address )
 5         $M68ARY ( Define the referencing code )
 6         2* M68ALLOT ( Update variable pointer ) ;
 7 FORTH DEFINITIONS
 8 ( n -- )
 9 : M68DARY ( Define a double precision array n cells long )
10         M68K M68PVAR @ ( Get base address )
11         $M68DARY ( Define the referencing code )
12         4 * M68ALLOT ( Update variable pointer ) ;
13 -->
14
15

Screen # 24

 0 ( M68K Cross Compiler -- Compiling words - arrays )
 1 M68K DEFINITIONS HEX
 2 ( adr -- )
 3 : $M68CARY ( Define code for a byte array )
 4         :M68MAC 30 C, 3C C, ( MOVE.W #const,D0 )
 5         $CON ( Compile address )
 6         D1 C, 56 C, ;M68MAC ;
 7 FORTH DEFINITIONS
 8 ( n -- )
 9 : M68CARY ( Define a byte array n cells long )
10         M68K M68PVAR @ ( Get base address )
11         $M68CARY ( Define the referencing code )
12         M68ALLOT ; ( Update variable pointer )
13 ;S
14
15


Screen # 25

 0 ( M68K Cross Compiler -- Control error checking )
 1 M68K DEFINITIONS HEX
 2 ( Error checking codes )
 3 1 CONSTANT $ECD-IF
 4 2 CONSTANT $ECD-BEGIN
 5 3 CONSTANT $ECD-DO
 6 4 CONSTANT $ECD-WHILE
 7 : $ERR-?PAIRS ( Abort if no control structure is started )
 8         M68?PAIRS @ 0=
 9         IF ." No control structure! " ABORT CR ENDIF ;
10 : $ERR-ABT ( Complete error message and abort )
11         ." expected " CR ABORT ;
12 : $ERR-IF ( Abort if no IF structure )
13         $ERR-?PAIRS $ECD-IF -
14         IF ." IF structure " $ERR-ABT ENDIF ;
15 -->

Screen # 26

 0 ( M68K Cross Compiler -- Control error checking )
 1 : $ERR-BEGIN ( Abort if no BEGIN structure )
 2         $ERR-?PAIRS $ECD-BEGIN -
 3         IF ." BEGIN structure " $ERR-ABT ENDIF ;
 4 : $ERR-DO ( Abort if no DO structure )
 5         $ERR-?PAIRS $ECD-DO -
 6         IF ." DO structure " $ERR-ABT ENDIF ;
 7 : $ERR-WHILE ( Abort if no WHILE structure )
 8         $ERR-?PAIRS $ECD-WHILE -
 9         IF ." WHILE structure " $ERR-ABT ENDIF ;
10 -->
11
12
13
```

(Continued on page 98)

Forth Compiler (Listing Continued, text begins on page 68)
Listing Two

```
14
15

Screen # 27

 0 ( M68K Cross Compiler -- Control structures )
 1 ( adr -- )
 2 : $FOR-RES ( Resolve a foreward branch )
 3         HERE OVER - ( Compute relative address )
 4         SWAP OVER HIGH-BYTE OVER C! ( Store high byte )
 5         1+ C! ; ( Store low byte )
 6 ( adr -- )
 7 : $BAK-RES ( Resolve a back branch )
 8         HERE - ( Compute relative address )
 9         $CON ; ( Compile address )
10 -->
11
12
13
14
15

Screen # 28

 0 ( M68K Cross Compiler -- Control structures )
 1 ( -- adr ecd )
 2 : IF     ( Compile IF structure, leave address to be resolved )
 3          ( and an error checking code )
 4          4A C, 5E C, 67 C, 00 C,
 5          HERE $ECD-IF 1 M68?PAIRS +!
 6          0 , ; ( Leave space for branch address )
 7 : ELSE   ( Compile an ELSE structure )
 8          $ERR-IF 60 C, 00 C,
 9          HERE SWAP ( Save current location and get IF adr )
10          0 , ( Leave space for branch address )
11          $FOR-RES ( Resolve IF branch )
12          $ECD-IF ;
13 : ENDIF ( Resolve an IF structure )
14          $ERR-IF $FOR-RES -1 M68?PAIRS +! ;
15 : THEN   ENDIF ;          -->

Screen # 29

 0 ( M68K Cross Compiler -- Control structures )
 1 ( -- adr ecd )
 2 : BEGIN ( Compile a BEGIN structure )
 3         HERE $ECD-BEGIN 1 M68?PAIRS +! ;
 4 : UNTIL ( Resolve BEGIN .. UNTIL loop )
 5         $ERR-BEGIN 4A C, 5E C, 67 C, 00 C,
 6         $BAK-RES ( Resolve BEGIN branch )
 7         -1 M68?PAIRS +! ;
 8 : AGAIN ( Resolve BEGIN .. AGAIN loop )
 9         $ERR-BEGIN 60 C, 00 C,
10         $BAK-RES ( Resolve BEGIN branch )
11         -1 M68?PAIRS +! ;
12 -->
13
14
15

Screen # 30

 0 ( M68K Cross Compiler -- Control structures )
 1 : WHILE ( Compile WHILE section of loop )
 2         DUP $ERR-BEGIN 4A C, 5E C, 67 C, 00 C,
 3         HERE $ECD-WHILE 0 , ; ( Leave space for address )
 4 : REPEAT ( Resolve BEGIN .. WHILE .. REPEAT loop )
```

```
 5          $ERR-WHILE SWAP $ERR-BEGIN
 6          60 C, 00 C, ( Code for back branch )
 7          SWAP $BAK-RES ( Resolve BEGIN branch )
 8          $FOR-RES ( Resolve WHILE branch )
 9          -1 M68?PAIRS +! ;
10 -->
11
12
13
14
15

Screen # 31

 0 ( M68K Cross Compiler -- Control structures )
 1 : DO ( Compile a DO structure )
 2          2F C, 1E C,
 3          HERE $ECD-DO 1 M68?PAIRS +! ;
 4 : LOOP ( Terminate a DO .. LOOP )
 5          $ERR-DO  52 C, 57 C, 4C C, 97 C, 00 C, 03 C,
 6          B0 C, 41 C, 6D C, 00 C,
 7          $BAK-RES ( Resolve DO branch )
 8          58 C, 8F C,  ( Drop index and limit )
 9          -1 M68?PAIRS +! ;
10 -->
11
12
13
14
15

Screen # 32

 0 ( M68K Cross Compiler -- Control structures )
 1 : +LOOP ( Terminate a DO .. +LOOP )
 2          $ERR-DO  30 C, 1E C, D1 C, 57 C, 4C C, 97 C,
 3          00 C, 06 C, 4A C, 40 C, 6E C, 04 C, B4 C, 41 C,
 4          60 C, 02 C, B2 C, 42 C, 6D C, 00 C,
 5          $BAK-RES ( Resolve DO branch )
 6          58 C, 8F C,  ( Drop index and limit )
 7          -1 M68?PAIRS +! ;
 8
 9 :M68MAC LEAVE 4 BYTES 3F 57 00 02 ;M68MAC
10 -->
11
12
13
14
15

Screen # 33

 0 ( M68K Cross Compiler -- Control structures )
 1 :M68MAC JSR.W 4 BYTES 30 5E 4E 90 ;M68MAC
 2 :M68MAC JSR.L 4 BYTES 20 5E 4E 90 ;M68MAC
 3 :M68MAC JMP.W 4 BYTES 30 5E 4E D0 ;M68MAC
 4 :M68MAC JMP.L 4 BYTES 20 5E 4E D0 ;M68MAC
 5 ;S
 6
 7
 8
 9
10
11
12
13
14
15

Screen # 34

 0 ( M68K Cross Compiler -- Initialization words )
 1 M68K DEFINITIONS HEX
 2 ( d -- )
 3 : A5LD  ( Load variable pool pointer )
 4          2A C, 7C C, $DCON ;
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
 5 : A6LD  ( Load data stack pointer )
 6         2C C, 7C C, $DCON ;
 7 : A7LD  ( Load return stack pointer )
 8         2E C, 7C C, $DCON ;
 9 -->
10 Note.. to create true modular programs there should be an
11 operating system that loads the appropriate registers and then
12 calls the module.  In that case these words should be discarded
13 since the address is determined at compile time instead of run
14 time.
15

Screen # 35

 0 ( M68K Cross Compiler -- Arithmetic words )
 1 HEX
 2 :M68MAC + 4 BYTES 30 1E D1 56 ;M68MAC
 3 :M68MAC - 4 BYTES 30 1E 91 56 ;M68MAC
 4 :M68MAC * 6 BYTES 30 1E C1 D6 3C 80 ;M68MAC
 5 :M68MAC / 8 BYTES 4C 9E 00 03 83 C0 3D 01 ;M68MAC
 6 :M68MAC D+ 4 BYTES 20 1E D1 96 ;M68MAC
 7 :M68MAC D- 4 BYTES 20 1E 91 96 ;M68MAC
 8 :M68MAC */ A BYTES 32 1E 30 1E C1 D6 81 C1 3C 80 ;M68MAC
 9 :M68MAC /MOD 8 BYTES 42 80 32 1E 30 1E 80 C1
10              4 BYTES 48 40 2D 00 ;M68MAC
11 :M68MAC MOD 8 BYTES 42 80 32 1E 30 1E 80 C1
12             4 BYTES 48 40 3D 00 ;M68MAC
13 :M68MAC */MOD 8 BYTES 32 1E 30 1E C0 DE 80 C1
14               4 BYTES 48 40 2D 00 ;M68MAC
15 -->

Screen # 36

 0 ( M68K Cross Compiler -- Arithmetic words )
 1 :M68MAC U* 6 BYTES 30 1E C0 DE 2D 00 ;M68MAC
 2 :M68MAC U/MOD A BYTES 32 1E 20 1E 80 C1 48 40 2D 00 ;M68MAC
 3 :M68MAC 1+ 2 BYTES 52 56 ;M68MAC
 4 :M68MAC 1- 2 BYTES 53 56 ;M68MAC
 5 :M68MAC 2+ 2 BYTES 54 56 ;M68MAC
 6 :M68MAC 2- 2 BYTES 55 56 ;M68MAC
 7 :M68MAC 2* 2 BYTES E1 D6 ;M68MAC
 8 :M68MAC 2/ 2 BYTES E0 D6 ;M68MAC
 9 :M68MAC NEGATE 2 BYTES 44 56 ;M68MAC
10 :M68MAC MINUS NEGATE ;M68MAC
11 :M68MAC DNEGATE 2 BYTES 44 96 ;M68MAC
12 :M68MAC DMINUS DNEGATE ;M68MAC
13 :M68MAC ABS 6 BYTES 4A 56 6C 02 44 56 ;M68MAC
14 :M68MAC DABS 6 BYTES 4A 96 6C 02 44 96 ;M68MAC
15 -->


Screen # 37

 0 ( M68K Cross Compiler -- Stack manipulation )
 1 :M68MAC DROP 2 BYTES 54 8E ;M68MAC
 2 :M68MAC 2DROP 2 BYTES 58 8E ;M68MAC
 3 :M68MAC DUP 2 BYTES 3D 16 ;M68MAC
 4 :M68MAC 2DUP 2 BYTES 2D 16 ;M68MAC
 5 :M68MAC SWAP 6 BYTES 20 16 48 40 2C 80 ;M68MAC
 6 :M68MAC 2SWAP A BYTES 20 16 2C AE 00 04 2D 40 00 04 ;M68MAC
 7 :M68MAC OVER 4 BYTES 3D 2E 00 02 ;M68MAC
 8 :M68MAC 2OVER 4 BYTES 2D 2E 00 04 ;M68MAC
 9 :M68MAC >R 2 BYTES 3F 1E ;M68MAC
10 :M68MAC R> 2 BYTES 3D 1F ;M68MAC
11 :M68MAC I 2 BYTES 3D 17 ;M68MAC
12 :M68MAC I' 4 BYTES 3D 2F 00 02 ;M68MAC
13 :M68MAC J 4 BYTES 3D 2F 00 04 ;M68MAC
14 -->
15
```

```
Screen # 38

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC = 6 BYTES 30 1E 32 1E B2 40
 2           8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
 3 :M68MAC < 6 BYTES 30 1E 32 1E B2 40
 4           8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
 5 :M68MAC > 6 BYTES 30 1E 32 1E B2 40
 6           8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
 7 :M68MAC MIN 6 BYTES 30 1E 32 16 B0 41
 8             6 BYTES 6F 02 C1 41 3C 80 ;M68MAC
 9 :M68MAC MAX 6 BYTES 30 1E 32 16 B0 41
10             6 BYTES 6C 02 C1 41 3C 80 ;M68MAC
11 -->
12
13
14
15

Screen # 39

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC D= 6 BYTES 20 1E 22 1E B2 80
 2            8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
 3 :M68MAC D< 6 BYTES 20 1E 22 1E B2 80
 4            8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
 5 :M68MAC D> 6 BYTES 20 1E 22 1E B2 80
 6            8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
 7 -->
 8
 9
10
11
12
13
14
15

Screen # 40

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC 0= 2 BYTES 4A 5E
 2            8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
 3 :M68MAC NOT 0= ;M68MAC
 4 :M68MAC 0< 2 BYTES 4A 5E
 5            8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
 6 :M68MAC 0> 2 BYTES 4A 5E
 7            8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
 8 :M68MAC D0= 2 BYTES 4A 9E
 9             8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
10 :M68MAC D0< 2 BYTES 4A 9E
11             8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
12 :M68MAC D0> 2 BYTES 4A 9E
13             8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
14 -->
15

Screen # 41

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC AND 4 BYTES 30 1E C1 56 ;M68MAC
 2 :M68MAC OR  4 BYTES 30 1E 81 56 ;M68MAC
 3 :M68MAC XOR 4 BYTES 30 1E B1 56 ;M68MAC
 4 :M68MAC 1'S 2 BYTES 46 56 ;M68MAC
 5 -->
 6
 7
 8
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
 9
10
11
12
13
14
15

Screen # 42

 0 ( M68K Cross Compiler -- Memory and I/O operations )
 1 :M68MAC ! 6 BYTES 30 1E 3B 9E 00 00 ;M68MAC
 2 :M68MAC @ 6 BYTES 30 16 3C B5 00 00 ;M68MAC
 3 :M68MAC 2! 6 BYTES 30 1E 2B 9E 00 00 ;M68MAC
 4 :M68MAC 2@ 6 BYTES 30 1E 2D 35 00 00 ;M68MAC
 5 :M68MAC +! 8 BYTES 30 1E 32 1E D3 75 00 00 ;M68MAC
 6 :M68MAC C! 8 BYTES 30 1E 32 1E 1B 81 00 00 ;M68MAC
 7 :M68MAC C@ A BYTES 30 16 42 41 12 35 00 00 3C 81 ;M68MAC
 8 :M68MAC FILL 8 BYTES 30 1E 32 1E 30 5E D1 CD
 9              8 BYTES 60 02 10 C0 51 C9 FF FC ;M68MAC
10 -->
11
12
13
14
15

Screen # 43

 0 ( M68K Cross Compiler -- Memory and I/O operations )
 1 :M68MAC AW! 4 BYTES 30 5E 30 9E ;M68MAC
 2 :M68MAC AW@ 4 BYTES 30 56 3C 90 ;M68MAC
 3 :M68MAC AL! 4 BYTES 20 5E 30 9E ;M68MAC
 4 :M68MAC AL@ 4 BYTES 20 5E 3D 10 ;M68MAC
 5 :M68MAC CAW! 6 BYTES 30 5E 30 1E 10 80 ;M68MAC
 6 :M68MAC CAW@ 8 BYTES 30 56 42 40 10 10 3C 80 ;M68MAC
 7 :M68MAC CAL! 6 BYTES 20 5E 30 1E 10 80 ;M68MAC
 8 :M68MAC CAL@ 8 BYTES 20 5E 42 40 10 10 3D 00 ;M68MAC
 9 :M68MAC 2AW! 4 BYTES 30 5E 20 9E ;M68MAC
10 :M68MAC 2AW@ 4 BYTES 30 5E 2D 10 ;M68MAC
11 :M68MAC 2AL! 4 BYTES 20 5E 20 9E ;M68MAC
12 :M68MAC 2AL@ 4 BYTES 20 56 2C 90 ;M68MAC
13 :M68MAC AFILL 8 BYTES 30 1E 32 1E 20 5E 60 02
14               6 BYTES 10 C0 51 C9 FF FC ;M68MAC
15 ;S

Screen # 44

 0 ( Definitions required for FORTH-79 )
 1 : <BUILDS CREATE SMUDGE 0 , ;
 2 : ENDIF [COMPILE] THEN ; IMMEDIATE
 3 ;S
 4 Note.. the above definitions work on my system which is a
 5 combination fig FORTH and FORTH-79.  However, the definition
 6 of <BUILDS may not work with a true FORTH-79 system.  The
 7 compatability depends on how the word DOES> operates in your
 8 system.  For the system described in Leo Brodie's book Starting
 9 FORTH the definition would be:
10 : <BUILDS CREATE ;
11 You will need to write an appropriate definition for your
12 system, the ones given above should serve as guides.
13
14
15
```

End Listing Two

Listing Three

```
Screen # 8

 0 ( Eratosthenes Sieve Prime Number program in FORTH )
 1 ( by Jim Gilbreath, BYTE September 1981 page 190 )
 2 FORTH DEFINITIONS DECIMAL
 3 8190 CONSTANT SIZE      0 VARIABLE FLAGS        SIZE ALLOT
 4
 5 : DO-PRIME       FLAGS SIZE 1 FILL
 6                  0 SIZE 0
 7                  DO FLAGS I + C@
 8                     IF I DUP + 3 + DUP I +
 9                          BEGIN DUP SIZE <
10                          WHILE 0 OVER FLAGS + C! OVER + REPEAT
11                          DROP DROP 1+
12                     THEN
13                  LOOP
14                  .  ." primes " ;
15 ;S

Screen # 9

 0 ( Eratosthenes Sieve Prime Number program for M68K compiler )
 1 ( Original by Jim Gilbreath, BYTE September 1981 page 190 )
 2 DECIMAL 0 M68CON #0 ( Note.. 0 is used a lot in the following )
 3 8190 CONSTANT SIZE      SIZE M68CON SIZE ( Both forms needed )
 4 M68VAR FLAGS    SIZE M68ALLOT
 5
 6 :M68MAC DO-PRIME FLAGS SIZE 1 LITERAL FILL
 7                  #0 SIZE #0
 8                  DO FLAGS I + C@
 9                     IF I DUP + 3 LITERAL + DUP I +
10                          BEGIN DUP SIZE <
11                          WHILE #0 OVER FLAGS + C! OVER + REPEAT
12                          DROP DROP 1+
13                     THEN
14                  LOOP ;M68MAC
15 -->

Screen # 10

 0 ( Test program to run the prime number program )
 1 HEX 7000. M68INIT ( Open the output file )
 2 :M68MAC INIT ( Initialize all the registers )
 3         800.    A5LD    ( Load variable pointer )
 4         4000.   A6LD    ( Load data stack pointer )
 5         7800.   A7LD    ( Load return stack pointer )
 6 ;M68MAC
 7 DECIMAL
 8 :M68K TEST ( Run the prime number test ten times )
 9         INIT
10         10 LITERAL #0 DO DO-PRIME DROP LOOP
11 ;M68K
12 M68END ( Close the output file )
13 ;S
14
15


Screen # 11

 0 ( Eratosthenes Sieve Prime Number program improved version )
 1 ( Original by Jim Gilbreath, BYTE September 1981 page 190 )
 2 DECIMAL 0 M68CON #0 ( Note.. 0 is used a lot in the following )
 3 8190 CONSTANT SIZE      SIZE M68CON SIZE ( Both forms needed )
 4 SIZE M68CARY FLAGS
 5
 6 :M68MAC DO-PRIME #0 FLAGS SIZE 1 LITERAL FILL
```

(Continued on page 106)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Three

```
 7                #0 SIZE #0
 8                DO I FLAGS C@
 9                   IF I 2* 3 LITERAL + DUP I +
10                        BEGIN DUP SIZE <
11                        WHILE #0 OVER FLAGS C! OVER + REPEAT
12                        2DROP 1+
13                   THEN
14                LOOP ;M68MAC
15 -->

Screen # 12

 0 ( Test program to run the prime number program )
 1 HEX 7000. M68INIT ( Open the output file )
 2 :M68MAC INIT ( Initialize all the registers )
 3         800.    A5LD    ( Load variable pointer )
 4         4000.   A6LD    ( Load data stack pointer )
 5         7800.   A7LD    ( Load return stack pointer )
 6 ;M68MAC
 7 DECIMAL
 8 :M68K TEST ( Run the prime number test ten times )
 9         INIT
10         10 LITERAL #0 DO DO-PRIME DROP LOOP
11 ;M68K
12 M68END ( Close the output file )
13 ;S
14
15
        .PROC   PRIME
;
;  Eratosthenes Sieve Prime Number program in M68000 assembly language.
;  This program provides a baseline for evaluating the performance of the
;  M68K compiler.
;
;  Register variables:
;       D0.. Temporary storage
;       D1.. Number of iterations
;       D2.. I - DO loop counter
;       D3.. P - candidate prime number
;       D4.. K - array index used with P
;       D5.. COUNT - number of primes
;       D6.. SIZE - size of the flags array
;
;       A0.. FLAGS - base address of the FLAGS array
;       A1.. Temporary address register used for initializing FLAGS
;
;
;  Note..
;       This program does not correspond exactly to the FORTH version but
;       the algorithm is the same so this should be a fair comparison.
;       The portions of the FORTH code which correspond to the sections of
;       assembler code are indicated in the comments.
;
;
SIZE    .EQU    8190
FLAGS   .EQU    800H            ;Base address of the FLAGS array
ITER    .EQU    10              ;Number of iterations of the sieve
;
        MOVE.W  #ITER,D1
        MOVE.W  #SIZE,D6
        MOVEA.W #FLAGS,A0
        BRA.S   ENDIL           ;Enter the iteration loop at the proper place
STARTIL                         ;Start of the iteration loop
;
;  FORTH code:
;       FLAGS SIZE 1 FILL
;
MOVEQ   #1,D0
MOVE.W  D6,D2           ;Load SIZE into D2
```

```
        MOVEA.L A0,A1           ;Address of element of FLAGS to set
        BRA.S   $02
$01     MOVE.B  D0,(A1)+
$02     DBRA    D2,$01
;
;  FORTH code:
;       0 SIZE 0 DO
;
        CLR.W   D5              ;Clear prime counter
        CLR.W   D2              ;Clear DO loop counter
DOLOOP
;
;  FORTH code:
;       FLAGS I + C@ IF
;
        BTST    #0,0(A0,D2.W)
        BEQ.S   THEN            ;If false, skip the true part of IF structure
;
;  FORTH code:
;       I DUP + 3 + DUP I +
;
        MOVE.W  D2,D3
        ADD.W   D3,D3
        ADDQ.W  #3,D3           ;P=I+I+3
        MOVE.W  D3,D4
        ADD.W   D2,D4           ;K=P+I
;
;  FORTH code:
;       BEGIN DUP SIZE <
;       WHILE 0 OVER FLAGS + C! OVER + REPEAT
;       DROP DROP 1+
;
BEGIN   CMP.W   D6,D4
        BGE.S   $03
        CLR.B   0(A0,D4.W)
        ADD.W   D3,D4
        BRA.S   BEGIN
$03     ADDQ.W  #1,D5           ;Update prime counter
;
;  FORTH code:
;          THEN
;       LOOP
;
THEN    ADDQ.W  #1,D2
        CMP.W   D6,D2
        BLT.S   DOLOOP
;
;
ENDIL   DBRA    D1,STARTIL      ;Repeat for requested number of iterations
        .END
```

End Listings

The FVG Standard Floating-Point Extension

by Ray Duncan
and Martin Tracy

[*Forth's development has been rather controversial, with even the nature of the language tending to work against standardization efforts. Perhaps because of the original fixed-point orientation of Forth, the movement to standardized floating-point operations for the language has lagged behind some of the standardization efforts of other high-level languages. The Forth Vendors Group, an association of vendors of Forth systems, has recently adopted its own standard for a floating-point extension to Forth. We present it here so that readers may examine what they may well be seeing in many of the Forth packages distributed in the future. If a large percentage of vendors choose to implement this, it may well have a significant impact in the next round of standardization by the Forth Standards Team. At the very least, it provides a look at one of the more recent attempts to bring more uniformity and portability to the language. — Ed.*]

In its recent release of the Forth-83 Standard, the Forth Standards Team concerned itself only with the core of the language. No attempt was made to introduce standardization in such areas as floating point, the interface to host operating systems, graphics, or file management. The Forth Vendors Group, hoping to help Forth find broader usage and acceptance, decided to take the initiative by creating and adopting the "FVG Standard Floating Point Extension." The document was written by Martin Tracy of Micromotion and Ray Duncan of Laboratory Microsystems, based on input from several vendors including Talbot Microsystems, Rising Star, Dysan, Miller Microsystems, and Creative Solutions.

The FVG Standard Floating Point Extension is intended to serve chiefly as a guide for implementors, and is a minimum functional description including command and function names and a few new data types. Although this FVG Standard Extension is not binding on any of the FVG members, we expect that most of the major vendors will be modifying their existing floating-point packages to conform with it. It will be submitted to the next meeting of the Forth Standards Team (expected to be in 1987) for consideration as part of the next Forth Standard. If Forth application programs that perform floating-point operations follow the guidelines laid out in the FVG Standard Floating Point Extension, referencing only names and functions described in that document, then such programs should then be portable without changes to the various floating-point implementations created by the different vendors.

"The FVG Standard is intended to serve chiefly as a guide for implementors, and is a minimum functional description . . ."

Ray Duncan, Laboratory Microsystems, P.O. Box 10430, Marina Del Rey, CA 90295.
Martin Tracy, Micromotion, 12077 Wilshire Blvd., Suite 506, Los Angeles, CA 90025.

The Forth Vendors Group Standard Floating-Point Extension

Syntax of Floating Point Numbers

If the floating-point extension word set has been overlaid onto an 83-Standard Forth system, a string of the following form will be compiled or interpreted as a real number:

nnnn.nnnExx

The "E" signifier is mandatory to force the conversion of a real number. The presence of numeric digits before or after the "E" is not required by this specification but may be mandatory in some implementations. A "–" sign may precede both the mantissa and the exponent, a leading "+" sign is also permissible on the exponent. A decimal point is optional and may occur anywhere in the mantissa. For example, all of the following numbers are legal:

–.0001E5
100.0E+0
1000.E–15

The word FNUMBER may be used by application programs to convert strings into real numbers. The current system BASE is assumed to be DECI MAL whenever floating-point numbers are being converted or displayed. If it is not, the results are undefined.

Display of Floating-Point Numbers

Two primitive display operations are always available: **F.**, which adjusts the position of the decimal point and prints the number without an exponent, and **E.**, which prints the number in exponential notation. More complex output formatting capabilities may be present in some systems and are described in the Optional Floating-Point Words glossary.

Floating-Point Stacks

Floating-point implementations differ in that some have a separate floating-point stack while others keep all integer and real numbers on the same parameter stack. This proposed extension does not concern itself with the relative benefits or disadvantages of either type. The source or destination of a real-number argument is always the floating-point stack, if separate stacks exist.

Portability

To ensure portability of an application, it is important that the programmer *not* take advantage of his knowledge of the physical format of real numbers. The floating-point variants of the memory access and stack operators should be used in all cases for real numbers, even when they happen to coincide with other Forth operators. For example, if the system supports a 32-bit real-number format, FDUP and 2DUP would have the same effect—but FDUP should be used for the sake of portability.

Floating-Point Glossary

The floating-point extension word set is described below in the standard glossary format. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes ("---") denote the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right. Symbols include:

addr	memory address
b	8-bit byte (upper 8 bits zero or ignored)
r,r1	floating-point number (length is implementation dependent)
f	boolean flag, 0=false, –1=true
n	16-bit, signed integer
u	16-bit, unsigned integer
d	32-bit, signed integer
du	32-bit, unsigned integer

Required Floating-Point Arithmetic Operators

F+ r1 r2 --- r3
Floating-point addition, leave the sum of r1 plus r2.

F– r1 r2 --- r3
Floating-point subtraction, leave the difference of r1 minus r2.

F* r1 r2 --- r3
Floating-point multiplication, leave the product of r1 and r2.

F/ r1 r2 --- r3
Floating-point division, leave the quotient of r1 divided by r2.

F** r1 r2 --- r3
Leave the value of r1 raised to the power r2, i.e. $r1^{r2}$.

FABS r --- [r]
Leave the *absolute* value of real number.

FNEGATE r --- –r
Change the sign of a real number.

FSQRT r1 --- r2
Floating-point square root.

FMAX r1 r2 --- rmax
Leave the larger of two real numbers.

FMIN r1 r2 --- rmin
Leave the smaller of two real numbers.

Required Floating-Point Transcendental Functions

FLOG r1 --- r2
Log to the base 10.

FLN r1 --- r2
Log to the base e.

FALOG r1 --- r2
Leave the value of 10 raised to the power of r1.

FALN r1 --- r2
Leave the value of e raised to the power of r1.

FSIN r1 --- r2
Leave the sine of r1. Input argument is in radians.

FCOS r1 --- r2
Leave the cosine of r1. Input argument is in radians.

FTAN r1 --- r2
Leave the tangent of r1. The tangent of $\pi/2$ or $3\pi/2$ radians returns the largest real number representable in the implementation's binary format.

FASIN r1 --- r2
Arc-sine, valid for $-1 <= r1 <= 1$.
Returns result in range $-\pi/2$ to $\pi/2$ radians.

FACOS r1 --- r2
Arc-cosine, valid for $-1 <= r1 <= 1$.
Returns result in range 0 to π radians.

FATAN r1 --- r2
Arc-tangent, valid for all real r1. Leaves result in range $-\pi/2$ to $\pi/2$ radians.

Required Floating-Point Logical Operators

The logical operators will regard two real numbers as equal if they differ only by a small amount. This "fuzz factor" is related to the magnitude of the real numbers and is implementation dependent.

F0= r --- f

True if floating-point number is equal to zero. The real number is removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

F0< r --- f

True if floating-point number is less than zero. The real number is removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

F= r1 r2 --- f

True if floating-point number r1 is equal to floating-point number r2. The real numbers are removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

F< r1 r2 --- f

True if floating-point number r1 is less than floating-point number r2. The real numbers are removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

Required Floating-Point Stack Operators

FDROP r ---

Discard floating-point number on top of stack.

FDUP r --- r r

Duplicate floating-point number on top of stack.

FOVER r1 r2 --- r1 r2 r1

Copy second floating-point number on stack to top of stack.

FSWAP r1 r2 --- r2 r1

Interchange two floating-point numbers on top of stack.

FROT r1 r2 r3 --- r2 r3 r1

Bring the third floating-point number to top of stack.

Required Real-Number Handling

FLOAT d --- r

Convert a signed double integer into its real-number equivalent, removing the double integer from the Forth parameter stack and leaving the result on the floating-point stack.

FIX r --- d

Convert a floating-point number to the nearest signed double integer equivalent, removing the real number from the floating-point stack and leaving the double integer result on the Forth parameter stack. The direction of rounding when a real number lies exactly halfway between two integers is implementation dependent. Underflow gives a zero result, overflow is an error condition and the value of the resulting double integer is undefined. Example: 2.7E0 FIX will return the double integer 3.

INT r --- d

Truncate a floating-point number to a signed double integer (round it toward zero), removing the real number from the floating-point stack and leaving the result on the Forth parameter stack. Underflow gives a zero result, overflow is an error condition. Example: 2.7E0 INT will return the double integer 2.

F! r addr ---

Store a floating-point number from the floating-point stack, at the address that is on the top of the Forth parameter stack.

F@ addr --- r

Fetch a floating-point number to the top of the floating-point stack, from the address that is on top of the Forth parameter stack.

FCONSTANT r --- (compilation)
--- r (execution)

A defining word used in the form:

r FCONSTANT cccc

When FCONSTANT is executed, it creates the definition cccc with its parameter field initialized to the value r. When cccc is later executed, the value r is left on top of the floating-point stack.

FVARIABLE --- (compilation)
--- addr (execution)

A defining word used in the form:

FVARIABLE cccc

When FVARIABLE is executed, it creates the definition cccc with its parameter field explicitly uninitialized. When cccc is later executed, the address of the parameter field is left on the stack, so that a F@ or F! operation may access this location.

Required Floating-Point Number Display

E. r ---

Display r in exponential form. The mantissa contains the maximum number of significant digits allowed by the floating-point data format, and the exponent is explicitly displayed even if it is zero. A trailing blank follows. If the current system base is not decimal, an error condition exists. For example, in the Micromotion implementation 12345.67E2 E. will display as .1234567E-07b (where each "b" character denotes an ASCII blank).

F. r ---

Display r on the currently selected output device in fixed-point form; i.e., the location of the decimal point is adjusted as necessary so that no exponent need be displayed. The number of digits specified by the most recent execution of the word PLACES (see below) are printed to the right of the decimal point. A trailing blank follows. For example, 4 PLACES 1.2345E02 F. will display as 123.4500b (where each "b" character denotes an ASCII blank).

PLACES n ---

Set the default number of digits that will be printed to the right of the decimal point by the F. operator.

Optional Floating-Point Commands

The Optional words have the same relationship to the proposed Floating-Point Extension as Controlled Reference Words have to the core 83-Standard; that is, if the function is available in the system, it should have the name and stack effect described below.

Optional Floating-Point Transcendental Operations

FSINH r1 --- r2

Hyperbolic sine.

FCOSH r1 --- r2

Hyperbolic cosine.

FTANH r1 --- r2

Hyperbolic tangent.

FASINH r1 --- r2

Inverse hyperbolic sine.

FACOSH r1 --- r2

Inverse hyperbolic cosine.

FATANH r1 --- r2

Inverse hyperbolic tangent.

PI --- r

Leave the real number π on the stack, represented with the maximum precision possible in the im-

plementation's binary floating-point format.

Optional Floating-Point Logical Operators

F0> r --- f

"True" if the floating-point number is greater than zero. The real number is removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

F> r1 r2 --- f

"True" if the floating-point number r1 is greater than the floating-point number r2. The real numbers are removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

Optional Floating-Point Display Words

Suggested error handling for floating-point display: if an overflow occurs during conversion for output, fill the display field with asterisks (the "*" character). If an underflow occurs during conversion, fill the display field

E.R r n1 n2 ---

Display r on the currently selected output device in exponential form with n1 digits to the right of the decimal point, right justified in a field of width n2. If the current system base is not decimal, an error condition exists. For example, 1.234E0 5 12 E.R will display as bb.12340E-01 (where each "b" character denotes an ASCII blank).

F.R r n1 n2 ---

Display r on the currently selected output device in fixed-point form with n1 digits to the right of the decimal place, right justified in a field of width n2. Numbers that cannot be represented within the given field width are printed in exponent form. If the current system base is not decimal, an error condition exists. For example, 1.2345E2 4 12 F.R will display as bbbb123.4500 (where each "b" character denotes an ASCII blank).

Optional Miscellaneous Real-Number Handling

FNUMBER addr --- r

Addr points to a counted string that is converted to a real number. The result is left on the floating-point stack. If the syntax of the string is not correct for a floating-point number, an error condition exists and the value of the result is undefined.

PACK d n --- r

Collapse a signed double-integer mantissa and a signed single-integer power of 10 into the corresponding real-number data format. Implementation dependent.

UNPACK r --- d n

Convert a real-number into a signed double integer corresponding to the mantissa and a signed single-integer power of 10. Implementation dependent.

F#BYTES --- n

Constant leaving the number of bytes in a real number as it is represented on the floating-point stack. Implementation dependent.

F#PLACES --- n

Constant leaving the maximum number of significant digits when a real number is converted to its decimal ASCII equivalent for display. Implementation dependent.

Optional Error Handling for Real-Number Operations

At minimum, the following error indicators should be available for inspection by the application program:

- overflow
- underflow
- division by zero
- output conversion overflow
- output conversion underflow
- attempted logarithm of zero
- attempted logarithm of number less than zero
- attempted square root of a number less than zero
- arcsin/arccos of argument outside the legal domain

FPSTAT --- addr

Addr points to an array of bits that contains error indicators for real-number operations. The number of flag bits and their significance is implementation dependent.

FPERR r1 n --- r2

Should be a deferred or vectored word to allow installation of altered error handling by the user. r1 is the result of a real-number operation and n indicates an error type. Sets an appropriate flag in FPSTAT and returns the real number r2, which may or may not be the same as r1, depending on the implementation and type of error.

Micromotion has written a Forth-83 implementation of this FVG Standard Floating-Point Extension. Machine code versions for the 6502, 8086, and 68000 are currently available. Micromotion may be contacted at 12077 Wilshire Blvd., Suite 506, Los Angeles, CA 90025.

DDJ

C/UNIX PROGRAMMER'S NOTEBOOK

by Anthony Skjellum

Past columns have been devoted to discussions about C and Unix as they are implemented in the real world. In this column, I depart from this precedent by discussing some proposals for increasing the power and flexibility of C. We work in an evolving field. Consequently, many languages go through regular upgrades and improvements (e.g., Fortran IV, Fortran 66 and Fortran 77). C has been more immune to change than other popular languages. The lack of upgrades to C is probably due mainly to its lack of intrinsic functions, but clearly points to the basic elegance and power of C.

I am aware of only a few upgrades to C beyond the language definition specified in *The C Programming Language* by Kernighan and Ritchie. These upgrades involved enumerated types and structure assignment. Both of these features were introduced with Unix version 7 and are detailed in the documentation for Unix version 7.

Structure assignment is useful in what follows, but I will not mention type enumeration. Therefore, although we base our definition of C on the Unix 7 version of C, this is not likely to confuse those who have access only to *The C Programming Language*.

Purists may argue that I have no business recommending changes or upgrades to C. Others may argue that many of the suggestions can be implemented via compiler preprocessors or by function calls and need not be part of the language. (I'll discuss this second point later in the column.) In order to head off the criticism that I am "tampering" with the C language, I offer my recommendations in the form of a new language grammar that is based on C but called X. I chose the letter *X* to denote language extensibility, which is the main point of the following proposals.

Language Extensibility

Most languages allow user-defined functions and subroutines, and many newer languages allow user-defined data tyes. Extensible langauges such as Forth and APL allow functions, operators, and data types to be added to the programming environment in a way that makes them equivalent in stature to predefined operations. Although C retains tremendous flexibility by excluding intrinsic functions, it does not allow user-defined types to be treated as easily as ints, longs or floats. Specifically, one cannot extend the definitions of operators such as Addition or Multiplication to new data types created with Typedef. This means that function calls must be used. Although this is a viable approach, it lacks elegance. This concept is illustrated in the following example.

As part of my C program, I need to define a data type called COMPLEX that will function like Fortran's complex data type. This data type is used for handling complex numbers of the form **A + B** where **i** is the imaginary unit and **A** and **B** are real numbers. This is shown with the definition in Figure 1 (page 116).

I will work with several variables of type COMPLEX (e.g., **alpha** and **beta**), which are defined as follows:

```
COMPLEX alpha, beta;
 /* alpha and beta are complex #'s */
```

Up to this point, we have treated the complex data type as we would built-in types. We can also work with pointers to or arrays of COMPLEX. However, to assign, add, multiply, or subtract these COMPLEX variables, we would have to invent subroutines. Subroutines for two representative operations are illustrated in Figure 2 (page 118).

The pseudocode presented with the subroutines in Figure 2 is the most convenient way to specify the operations desired. If the data types had been intrinsic, we could have used similar real C statements in lieu of subroutines. To utilize +, * or other operators with the COMPLEX data type, we must introduce a mechanism for defining these operations.

Operators

How could we specify new operations? For example, how would we define Addition for the complex data type? Figure 3 (page 120) shows the type of definition that could be used to extend Addition to the COMPLEX type.

The keyword **oper** is new: oper indicates that the definition following it is for an operator. The **return** keyword used in function calls also appears with a similar meaning. Because COMPLEX preceeds oper, this example defines an operation over the COMPLEX data type. Because there are two arguments (alpha, beta), the operator is binary. Finally, note that the + sign is enclosed in graven accents. Quoting by graven accents is chosen as a way to distinguish operator names. We will see that quota-

```
typedef struct /* complex number type definition */
{
                                    double _creal ; /* real part */
                                    double _cimag ; /* imaginary part */
} COMPLEX
```

Figure 1

tion will not always be needed.

To use this new operator—and assuming that = had also been defined—the following statement could be used:

```
gamma = alpha + beta;
        /* add complex numbers */
```

Note that we have omitted the graven accents. Because the + can be distinguished from keywords or identifiers in this context, quoting is not required.

The operator definition specified above gives the X compiler a means to evaluate the Addition request specified in the example statement. The parser would break this statement down until it could pass an argument garnered from the left and right of the Addition operator, much as it does with intrinsic operators and data types. Whether this results in a subroutine call or in-line code would depend on the compiler's implementation.

More on Operators

Operators are a powerful and useful concept. We needn't limit ourselves to defining standard operations for new types. There is nothing to stop the definition of arbitrary operators. A crude facility already exists for this in C via the parameterized #define statement. However, facility just discussed is more general and more consistent with the syntax of C than the preprocessor #define approach. To encompass the generation of in-line code as provided by #define, we would also offer the inline adjective, which could be used as follows:

```
COMPLEX inline oper
    `-`(alpha, beta)
                /* subtraction inline */
```

This keyword would instruct the compiler to generate in-line code (as opposed to a subroutine call) whenever possible. Its use is analogous to the use of the register adjective: the compiler complies when feasible and ignores the request when it cannot comply.

In some cases, C definitions can be shortened when no ambiguity exists (e.g., "unsigned" instead of "unsigned int"). Therefore, "inline" would replace "inline oper" in actual practice. Furthermore, operators would, by default, work on and return integers, as functions do by default.

Other Uses for Operators

In my view, operators would be used not only to define existing operations over new data types, but also for specifying other operations over new as well as existing data types. These new operators would normally have alphanumeric names and would thus require quoting in graven accents when they appear in expressions. For example, we define the operation of NAND (negated and) for integers as follows (no graven accents are required in the definition, but they are required in the invocation below):

```
int oper nand(a,b)
int a,b;
{
    return(~(a & b));
}
```

To use this in an actual expression, we would have to quote the nand:

```
c = a `nand` b;
```

Operator Hierarchy

C already has a built-in hierarchy for known operations. The most reasonable approach is to give user-defined operators the lowest priority. This might require more parentheses, but seems logical.

Pointers to Operators

C provides the facility to use pointers to functions. It could potentially prove useful to have pointers to operators as well. You specify a function's address by its name without trailing parentheses. Unfortunately, operator names are used in this way to indicate the operation they represent. In order to remove the ambiguity in requesting the pointer, the operator name could be parenthesized—for example; (+) or (`nand`).

Using pointers to operators implies that defined operations must have subroutines associated with them. Thus, truly in-line operators could have no pointers associated with them.

Dichotomy of Operators and Functions

Functions and operators are almost the same thing. However, the compiler must know if an operator is binary or unary. Therefore, the operator definition must be available before you use it. On the other hand, arguments to C functions are not checked for number or type. Therefore, we choose to keep operators and functions separate, although there is nothing to prevent operators from using function calls.

In order to avoid lexical conflicts, operator and function names would have to be different. This is also desirable from a programming viewpoint, in order to avoid confusion and errors.

Other Proposals

With the addition of operators, the X grammar provides a much more consistent programming environment than standard C. However, there are some other points that deserve consideration. The first of these is the need to provide a means to handle subroutines with a variable number of arguments.

Because C makes no assumptions about its function library, the user is free to write his own library, should the standard functions prove inadequate. However, the user cannot properly handle functions that have variable numbers of arguments, as must be done by printf( ), scanf( ), and their relatives. We solve this problem by introducing a typing adjective called **vec** (vector). This adjective is used to indicate that the number of arguments to the function is variable. For example, the fictitious function my_printf( ), which allows variable arguments (and returns an integer), would be defined as follows:

```
vec int my_printf(argcnt,argvec)
int argcnt;
char *argvec[ ];
{
                /* code goes here */
}
```

A function declared with vec always has two arguments: argcnt and argvec. These variables are analogous to main( )'s (argc,argv) pair. Before use, a definition of the form:

```
vec my_printf( );
```

would be included in each file where my_print( ) is referenced. This definition causes command line arguments to be processed normally; the rightmost argument is pushed (placed on the stack) first and the leftmost last

when code is generated. However, the two additional arguments argcnt and argvec are also stacked. The argvec variable points to the stack location where the first real argument is located. Since normal stacks are pushdown, this should provide the arguments in the correct order—argcnt contains the number of arguments plus one to account for argvec. This makes it completely analogous to argc. The argument argvec always contains an address, but this is not very useful if no arguments were specified in the function call.

To illustrate the stacking mechanism, imagine that we invoke my_printf( ) as follows:

```
my_printf( )(arg1,arg3,arg4,arg5) ;
```

For this specific call, the stacking arrangement (excluding any special register saves) woud look as specified in Figure 4. Note that argcnt is six (five arguments) for this case, as described above.

It might be worthwhile to have variable argument calls, even if the function were not declared as using this calling convention. To allow this, we introduce the ellipsis (. . .) concept into the argument string. If my_printf( ) were not declared as vec, we could force a variable argument format as follows:

```
my_printf( )
    (arg1,arg2,arg3,arg4,arg5 . . . );
```

Including the ellipsis mark for this variety of call seems to improve readability, but is not required for compatibility with current C usage.

Fixed Arguments

The argvec variable always points to the first variable specified in the function call. However, the function definition could still explicitly declare a finite number of arguments that it may wish to examine directly. For example, if the first argument of my_printf( ) were a control string, we could declare my_printf( ) as shown in Figure 5.

Notice that the contents of control_string would be meaningless if argcnt were less than two.

One final note about variable argument control: it enhances a function's abiliity to detect incorrect input. With reference to printf( ), Kernighan and Ritchie state: "A warning: printf uses its first argument to decide how many arguments follow and what their types are. It will get confused if there are not enough arguments or if they are the wrong type."

If implemented with the vec arrangement, printf( ) could at least know if it has been given the right number of arguments. It still would not know if they were of the correct types.

Variable-length Automatic Arrays

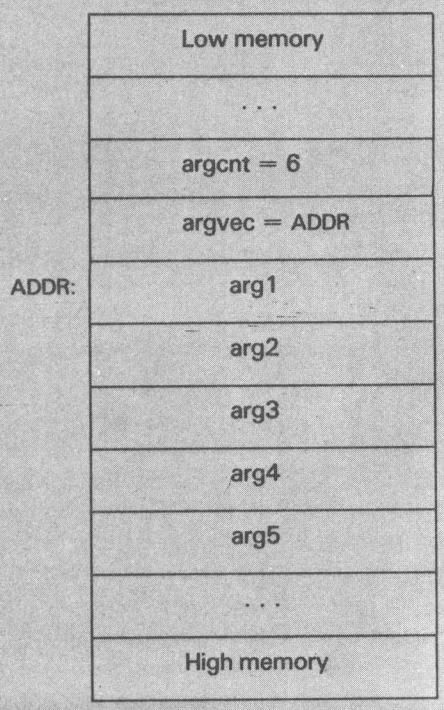
Another element of the X grammar is its ability to declare automatic arrays that possess variable length. Because stack displacements are computed at

```
Assignment: alpha = A + iB; /* pseudo code */
Function:
      calling sequence     (K&R C):        cassign(&alpha,A,B);
      calling sequence     (Unix 7 C):     alpha = cassign(A,B);
      function definition  (K&R C):
            cassign(comp,a,b)
            COMPLEX *comp;
            double a,b;
            {
                  comp->_creal = a;
                  comp->_cimag = b;
            }
      function definition  (Unix 7 C);
            COMPLEX cassign(a,b)
            double a,b;
            {
                  COMPLEX temp; /* temporary variable */
                  temp._creal = a;
                  temp._cimag = b;
                  return(temp); /* return structure */
                  }
Addition: gamma = alpha + beta; /* pseudo code */
Function:
      calling sequence     (K&R C):        cadd(&gamma,&alpha,&beta);
      calling sequence     (Unix 7 C):     gamma = cadd(alpha,beta);
      function definition  (K&R C):
            cadd(gamma,alpha,beta)
            COMPLEX *gamma; /* destination */
            COMPLEX *alpha; /* addend */
            COMPLEX *beta; /* augend */
            {
                  gamma->_creal = alpha->_creal + beta->_creal
                  gamma->_cimag = alpha->_cimag + beta->_cimag
            }
      function definition  (Unix 7 C);
            COMPLEX cadd(alpha,beta)
            COMPLEX alpha,beta; /* addend, augend */
            {
                  COMPLEX temp; /* temporary */
                  temp._creal = alpha._creal + beta._creal;
                  temp._cimag = alpha._cimag + beta._cimag;
                  return(temp);
            }
```

Figure 2

```
COMPLEX oper `+`(alpha,beta) /* X
grammar */
COMPLEX alpha,beta ;
{
    COMPLEX __temp; /* temporary
    */
    __temp.__creal = alpha.__creal + beta.__creal ;
    __temp.__cimag = alpha.__cimag + beta.__cimag ;
    return(__temp) ; /* return result */
```

Figure 3



Memory Layout for a variable argument function call

Figure 4

```
vec int my_printf(argcnt,argvec,control_string) ;
int argcnt ;
char **argvec ;
char *control_string ;
```

Figure 5

each entry to a block, a computed size allocation is forced. At worst, a memory allocation mechanism must be tied into the compiler. This latter restriction can be serious if C is used in a very-low-level environment, such as in operating system development. So that the use of this feature can be seen readily, we require the use of the var adjective in conjunction with such definitions. For most purposes, the feature is a welcome enhancement. Where it is inappropriate, this feature should be disabled via a compiler switch.

As a general example, we declare a variable-length array in the routine in Figure 6 (page 122).

A New Looping Structure

Many loops are unconditional, with breaks generated only from within. Therefore, it is often useful to have an unconditional looping command. This avoids a lot of while(1) sequences. The command could be implemented as follows:

```
    loop
    {
                                    . . . code . . .
    }
replaces
    while(1)
    {
                                    . . . code . . .
    }
```

Preprocessors and Related Comments

Preprocessors could be used to implement several of the X features mentioned above. The statements and expressions would be expanded by the preprocessor into standard function calls. The preprocessor would also provide subroutines from definitions as needed. New data types could certainly be handled in this way. However, changes to the C parser must be made in order to handle the vec and var features. Trivial additions such as loop can be handled with the existing C preprocessor.

Some programmers may argue that no additions are needed, because most of the features outlined above can all be achieved through function calls. In my view, the X grammar makes C

(Continued on page 128)

OF INTEREST

by Michael Wiesenberg

FIG Member Services

The **Forth Interest Group (FIG)** wants you to join up, and its newly elected board is emphasizing member services as one of its prime objectives. The services of this nonprofit, member-supported organization of more than 4800 members and 53 chapters worldwide include publication of *FORTH Dimensions*, the FIG-Tree (on-line computer data base), a job registry, various conferences, user library, membership directory, speaker bureau, catalog of software, educational tapes, and the annual Forth Convention (November 16–17 in Palo Alto, California). You get it all for $15 a year ($27 foreign).

Four Times the Forth

FourByteForth, from Software Architects, is a 32-bit Forth for 68K machines. It runs under CP/M-68K or boots directly on the Sage Computer. Your $250 gets you a full-screen editor, full-screen debugger, interpreter, compiler, decompiler, assembler, and user manual. It's supposed to be fast; the company claims it compiles 125 screens per minute.

More Freeware

I'm highly in favor of programs you get for the cost of a diskette—or free if you supply diskette and postage—and then pay for if you like. Charter Software offers **PC - General Ledger** for the IBM PC and PC compatibles. The program is a cash-receipts journal, check register, journal entry, trial balance, and financial-statement system that also prints checks. If you like it, you send in $50, and then you also get enhancements, updates, phone support, and the manual.

Fortran 77 Library

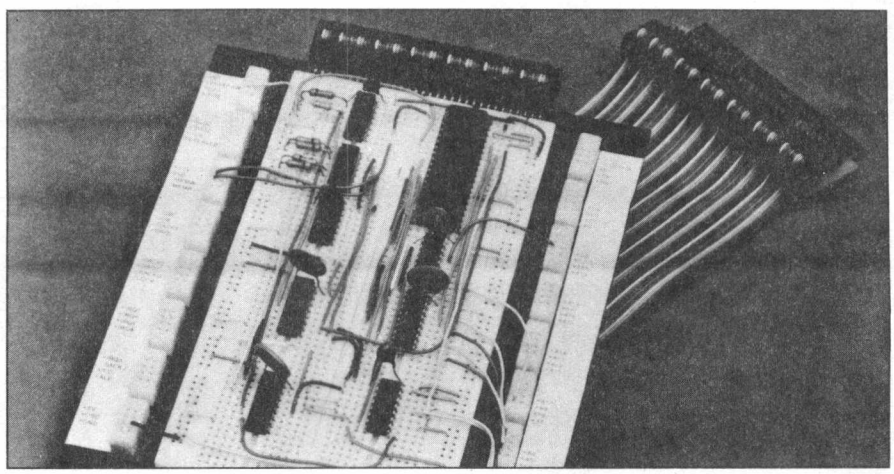
Peerless Engineering Service offers a **Fortran 77 Scientific Subroutine Library** for MSDOS and PCDOS, using Microsoft Fortran 3.13, with 114 scientific subroutines, from statistical to numerical analysis. It includes solutions to third-order differential equations, solutions to m equations in n unknowns, and various approximations. Need most flavors of logarithms, including operations on complex numbers? Multiple and linear regressions? Fourier analysis? All here. Order directly for $49.95 through the firm's toll-free number.

More for C64

Practicorp offers several products of interest to Commodore 64 owners. **PractiFile** is a data base program that handles lists and numbers. The program is designed to handle inventory control, mailing lists, accounts receivable and teachers' grade books; it also monitors charge accounts and bank balances and prints mailing labels or

Breadboard It Yourself

eZ BOARD, from Sabadia Export, provides breadboardng for the masses—a glass epoxy PCB with solderless breadboarding units for building circuits—that allows you to make your own experimental add-ons for personal computers. The product contains four separate distribution buses with 50 tie points each. You get 2090 tie points with a capacity of 24, 14-pin DIP switches. The company says you simply plug in components with lead diameters up to .032 and connect them with ordinary, solid hookup wire. (The firm didn't say .032 *what*. Do you suppose it meant inches? Or maybe .032 angstroms?) Models IPC, APC, and CPC are available for IBM PC, Apple, and Commodore systems (and all hardware-compatible computers of each type), and more are coming. Send $174.95 plus $5 shipping to get board, cable, and connectors.



statistical reports. It interacts with other Practicorp software products, plus various word-processing programs. The company claims Practi-File is easy to use, that its ability to handle more than 1000 mailing-list records exceeds the capacity of other DBMS programs for the C64. Fields can be of random size, and maximum record size is 254 characters. Sorts are fast, and you can specify up to five nested keys. You can have multiple lines per record and use arithmetic functions on columns. The disk costs $55.

64 Doctor is a diagnostic program that looks at the system's hardware and pinpoints malfunctions involving keyboard, audio, video, joystick, RS-232 port, disk drive, printer, RAM, or cassette player. It's $29.95 on disk and $24.95 on tape.

Insured Against Infringement

This month's unusual claim is that **IBM PC-compatible ROM BIOS** software from Phoenix is insured against infringement suits. Just think how much better Franklin's financial position would have been had it insured its operating system. If you're an OEM who needs unlimited licensing on this product, $290,000 will do the trick. I like to let *DDJ* readers in on these little bargains.

Word Processing on the C64

Commodore 64 owners sometimes sneer knowingly when someone talks of word processing on the C64; those sceptics cite the machine's limitation of file size to a maximum of about seven pages and its inability to display 80 characters without an 80-column board. Muse Software claims its **Super-Text** addresses those and other problems. For $99, you get unlimited document size, multiple-file search and replace, split-screen abilitiy, 80-column display without additional hardware, on-screen formatting, on-screen help, word wrap, and single-key commands. The product is also available for Atari 400/800/1200XL computers, but without the split-screen feature or 80-column ability. An IBM PC version of Super-Text has just been published. The original version, for the Apple II and IIe, is a perennial bestseller. The Apple and IBM PC versions each cost $175.

Old Fortrans for New Cs

Don't throw away your old Fortran code just because you're updating to a trendy new C environment. **FORTRIX**, from Rapitech Systems, converts Fortran programs and files to maximally portable C code. The new C code retains the essence of the original Fortran source, fully structured and parallel to the original, yet in functionally equivalent C statements. It costs $2500, configured, the firm says, for "various" Unix environments.

T_EX and Tingler! on 3000

T_EXies will be pleased to hear they can now get **T_EX** for their HP-3000s, from **TeXeT**, and produce flawless, typeset-quality documents. They can enter those documents from any terminal and print them on HP laser printers. More than 50 fonts are available, in point sizes from 5 to 72, with special mathematical and language symbols. The license fee is $1600, including a software device driver for a daisy-wheel printer; you'll pay $1600 more for the laser printer driver.

Tingler!, from TeXeT, gives information on run-time behavior of programs on the HP-3000. You can optimize and resegment programs based on this information; the information includes how many times each procedure and system intrinsic was called and where it was called from, how many intersegment transfers there were and where they occurred, and how much CPU time was expended in each procedure and intrinsic. Tingler! also detects logic errors and redundant code. It analyzes the program by patching itself into a copy of the executable program file, collecting performance data during a normal run of the program, and providing an off-line analysis—all without requiring recompilation of the program. License fee is $1000.

Go East, Young Programmer

One of the most interesting computer conferences of the year is likely to be **Computer China**, to be held in Xiamen Special Economic Zone in the Peoples' Republic of China from November 25 to December 1. This event is sanctioned by the Ministry of Electronics and backed by the China Microcomputer Applications Association, the China Research Society for Modernization of Management, and the China Instrument Society.

Another interesting conference, also to be held in the Orient, is billing itself as the first to be held on a ship. The **AFIPS-ASIA '85**—"The First Floating Shipboard Computer Expo"—is sponsored by the American Federation of Information Processing Societies (AFIPS). The ship, the *World Wide Expo*, has two large exhibition halls, five conference rooms, several international restaurants, and private businessrooms. It leaves Tokyo on February 14, 1985, stops at Osaka and Kitakyushu (Japan), Taipei, and Hong Kong, and docks in Singapore on March 2. AFIPS, by the way, is comprised of the folks who bring you NCC and the Office Automation Conference (OAC).

IEEE Confers with ACM

Allow me to be the first to announce that the world's two largest professional societies for computer scientists and engineers, the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE), will hold their first joint annual technical conference. Scheduled to be held in November, 1986, the

gathering is to be called the **Fall Joint Computer Conference (FJCC)**. The two organizations have combined membership ofmore than 125,000.

Contact Points

Association for Computing Machinery (ACM), 11 West 42nd St., New York, NY 10036; (212) 869-7440.

American Federation of Information Processing Societies (AFIPS), 1899 Preston White Drive, Reston, VA 22091; (703) 620-8900.

Charter Software, Box 70, Monticello, IL 61856; (217) 762-7143.

Computer China, c/o Kallman Associates, 5 Maple Court, Ridgewood, NJ 07450; (201) 652-7070.

Forth Interest Group, Box 1105, San Carlos, CA 94070; FIG Hot Line: (415) 962-8653.

Institute of Electrical and Electronics Engineers Computer Society, 1109 Spring St., Suite 300, Silver Spring, MD 20901; (301) 589-8142.

Muse Software, 347 North Charles Street, Baltimore, MD 21201; (301) 659-7212.

Peerless Engineering Service, 5819 Soquel Drive, Soquel, CA 95073; (408) 462-0330, or for orders, call (800) 824-7888, Operator 419.

Phoenix Software Associates, Ltd., 1420 Providence Highway, Suite 260, Norwood, MA 02062; (617) 769-7020.

Practicorp International Inc., The Silk Mill, 44 Oak Street, Newton Upper Falls, MA 02164, (617) 965-9870.

Rapitech Systems, Inc., 565 Fifth Ave., New York, NY 10017; (212) 687-6255.

Sabadia Export Corp., Box 1132, Yorba Linda, CA 92686; (714) 630-9335.

Software Architects, 1912 Grant, Berkeley, CA 94703; (415) 549-3185.

TeXet Company, 163 Linden Lane, Mill Valley, CA 94941; (415) 388-8853.

(Continued from page 120)

more (and not less) consistent because it allows intrinsic and user-defined types to be handled similarly. It also allows greater portability by defining a means through which variable argument functions can be handled uniformly. In summation, it turns C into an extensible language while adding only a few new keywords.

Conclusion

In this column, I have suggested an enhanced C grammar, which was denoted X to indicate extensibility. It is the (Unix 7) C language with enhancements designed to allow the incorporation of user-specified operators into programs. This should provide more flexible and consistent reference to user-defined data types. Also mentioned were variable-length automatic arrays (var) and a mechanism for allowing variable argument functions (vec). Finally, the use of preprocessors for implementing these ideas was mentioned.

I look forward to any other ideas about X or enhanced C that may be forthcoming from readers. **DDJ**

```
/* declare an array of integers one larger than argument */
array_test(length)
int length ;
{
    var int test[length+1]; /* declare array */
    . . .
}
```

Figure 6

GREP.C

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

#96 October 1984

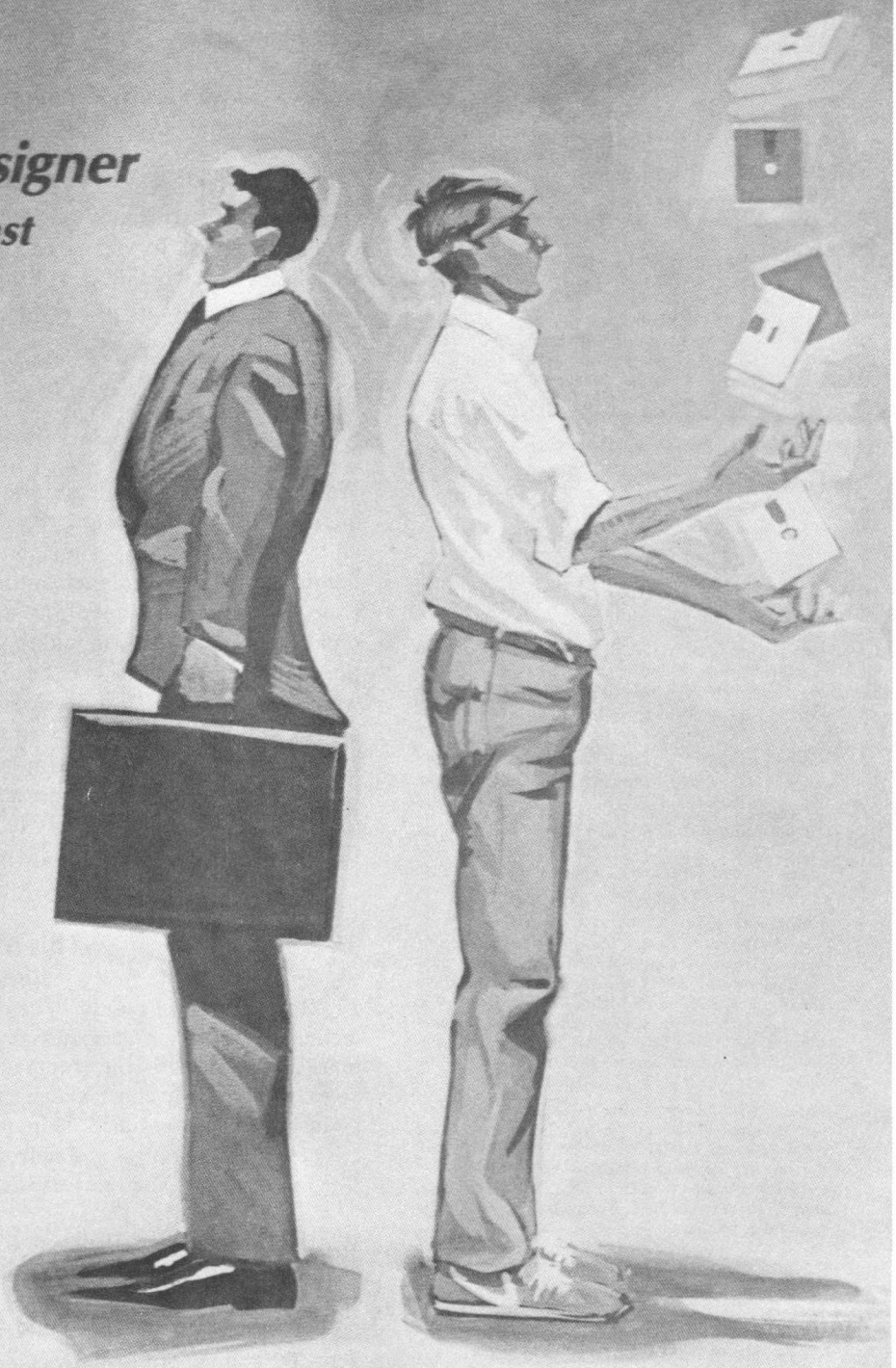
$2.95 (3.50 Canada)

New Column: The Software Designer

East Coast vs West Coast Programming Style

Simple Calculations With Complex Numbers

More dBASE II Programming Techniques



0 10 38351 16562

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*
Editorial Intern *Mark Johnson*

Production
Design Director *Fred Fehlau*
Art Director *Shelley Rae Doeden*
Production Manager *Detta Penna*
Production Assistant *Alida Hinton*
Cover *William Cone*

Advertising
Advertising Sales *Alice Hinton, Walter Andrzejewski*

Circulation
Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

OCTOBER 1984
VOLUME 9, ISSUE 10

CONTENTS

This Month

If you have not already noticed, we have a new column starting this month. Mike Swaine will be writing The Software Designer, a column intended to present the views of industry leaders on software design topics. This debut discusses issues of programming style differences. Tools for software design will be next month's topic.

Also as of this issue, Of Interest will be written by Randy Sutherland. Our congratulations to Randy on his new endeavor. Michael Wiesenberg, who has handled that task for nearly two and a half years, will be moving on to do other writing for us, including one of the new puzzle columns we have coming up. Our thanks to Michael for the job he's done for us. We look forward to his future contributions.

Coming Down the Pike

We've been looking at our editorial calendar for 1985 and thought we'd let you in on some of the special issues we've planned. As you already know, we plan to focus on Unix this December. The February 1985 issue will be our 100th issue, and we figure we should do some celebrating. March 1985 will be an artificial intelligence issue, including the winner from the Fifth Generation Programming Competition (you did get your entry in, didn't you?). June 1985 will focus on telecommunications. September will still be the annual Forth issue, and we will close out the year with a focus on the latest developments in operating systems (the exact topics, of course, partially dictated by what transpires during the year).

While this is only part of what we have in store, it will give readers something to look forward to and authors something to plan toward. Authors should note that the copy deadline is likely to be about three months prior to the issue date, though we will announce deadlines as we get closer to the issues.

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to remarks to the editors concerning accuracy and relevance of manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness. Their remarks help prevent authors form exposing blindspots or misconceptions in print and help ensure that our readers receive clear and accurate information. The referees who contributed to this month's issue are:

Wayne Chin, Hewlett-Packard, Information Networks Division
Georges Grinstein, Computer Science Dept., Fitchburg State College
John P. Keyes, Microsoft
Ben Laws, Computer Science, North Texas State University
Scott D. Thomas, Informatics General Corporation

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

More dBASE II Programming Techniques 26
by Gene Head
The explanation of several undocumented dBASE II features includes SET CALL TO ADDRESS and CALL VARIABLE, as well as the dBASE hex file format.

Simple Calculations with Complex Numbers 30
by David D. Clark
The author discusses the theory of complex numbers and complex functions and how they can be implemented in computer programs.

GREP.C—A Unix-like Generalized Regular Expression Parser 50
by Allen Holub
Born of the need for something more general than the pattern-searching capability of a text editor, this pattern matcher implements most of the features of the Unix utility of the same name.

DEPARTMENTS

Editorial 6

Letters 10

Dr. Dobb's Clinic 14
Putting his money where his mouth is, the Intern presents an optimization scheme for compilers on microcomputers

CP/M Exchange 20
One of those real-life embarrassment stories . . .

The Software Designer 22
What's a West Coast programmer? Our new column on software design issues begins with a roundtable discussion of programming style

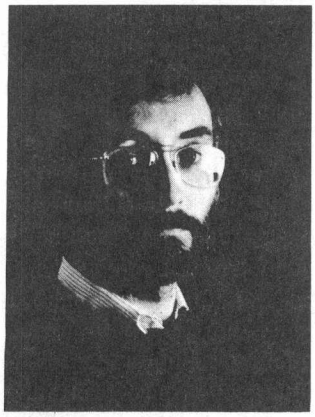
16-Bit Software Toolbox 84
MSDOS 2.0 Filters, Sizing RAM under MSDOS, More Microsoft Assembler Warnings, Hex to ASCII Conversion

C/Unix Programmer's Notebook 92
A set of routines to simplify user-program interaction, and two programming systems illustrating Runge-Kutta integration

Book Reviews 118

Of Interest 126

EDITORIAL



Five months ago I wrote my first editorial for *DDJ* and in it made several promises about the editorial direction of the magazine. Since then, editor Reynold Wiggins and I have spent twenty-odd weeks examining the Doctor, and recently the entire *DDJ* staff spent a three-day weekend poking *DDJ* with every available instrument. The prognosis is excellent; *DDJ* is a healthy, growing magazine, and I can now be more specific about what the magazine will not be doing over the next year.

DDJ will not abandon 8-bit software so long as useful creative software development is being done on 8-bit systems; forever, so far as we can tell. *DDJ* will, on the other hand, focus more deliberate attention on 16- and 32-bit software. We'll publish MSDOS- and Unix-based software and we'll investigate software development on the newer microprocessors. If you conclude that we intend to stay pretty loose, you're right.

DDJ will not become a product-specific magazine. We'll always prefer the MSDOS application to the PCDOS and the DOS-independent application to the strictly MSDOS. We are interested in the Mac/Lisa/PARC human interface, naturally; it's a highly significant development in personal computer software. But the insights embodied in the Mac are not proprietary.

DDJ will not become language-specific, either. Most of the code published in other computer magazines is in BASIC. Not so in *DDJ*, since we try to furnish consequential code for software designers. Despite the fact that this magazine was founded to put a version of BASIC in the public domain, its editors have never considered BASIC an ideal language either for software development or for communicating ideas about programming.

Apparently Kemeney and Kurtz agree; the designers of BASIC seem to have concluded that BASIC would be better off being Pascal (i.e. True BASIC) just as Pascal's designer declares his language obsolete, to be succeeded by Modula-2. Pascal remains, however, a good notation for communicating algorithms, so we'll doubtless continue to publish some Pascal code. We'll be investigating Modula-2, but we do have some questions about how and in what settings it's likely to be useful (see "The Software Designer" in this issue). Our Modula coverage is likely to focus at least for the immediate future on what the language has to offer programmers (as in "An Introduction to Modula-2 for Pascal Programmers," May 1984), rather than on providing tools for Modula programmers.

DDJ has participated in the evolution of Forth as a language, but we will shift our coverage of Forth slightly over the next few months. We disagree with *Byte* magazine's assessment that Forth is still a language in flux. Such developments as the FVG Forth floating-point standard (described in *DDJ* last month) are evidence that Forth is ending its formative years and is becoming an adult language. That being our view, we'll be leaving some of the residual wrangling to other, language-specific forums, and we'll concentrate on providing useful Forth tools.

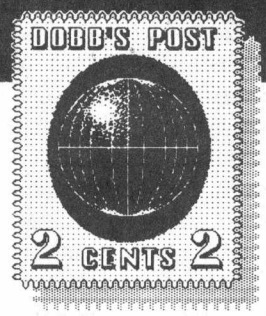
The most important language in microcomputer software development today is C, and *DDJ* will continue in 1985 to be the best source for new programming tools in C.

Finally, *DDJ* will not lower its technical standards. Many readers have insisted that we not water down the technical level of the magazine. Heaven help us if that's bad advice, because we're going to follow it. On the other hand, don't be alarmed to see a page here and there given to humor, puzzles or programming competitions. Serious doesn't mean stoic.

Michael Swaine

Michael Swaine

LETTERS



Commodore Commentary

Dear *DDJ*:
The June 84 article on CP/M on the Commodore 64 is "just what the doctor ordered." I think that the Commodore will become a very important tool, but there is one fly in the ointment. The disk drive is s-l-o-w. A serial data link is not as stiff or noisy as flat cable (the dreaded RFI monster) but is inherently slower. Examination of the *Commodore 64 Programmers Reference Manual* seems to indicate something seriously wrong. It just couldn't be *that* slow. Even with 1000 picofarad loading, the 1000 ohm pull-up resistor would give a fast 1 microsecond time constant. Data handling in the CPU is the real problem, but even so, it should take less than 30 microseconds per bit. A 256-byte disk block should transfer in less than 62 milliseconds. It may be that cycle stealing by the display device requires huge operating margins. In this case, letting the CPU provide data clocking for both incoming and outgoing block transfers eliminates the need.

I may have missed something obvious, but that still does not excuse Commodore's "floppy tape" philosophy.

Yours:
Frank J. Wilson
2468 Elden St., Apt. J
Costa Mesa, CA 92627

More Chinese Forth

Dear Doctor:
As a longtime student of Chinese and a recent convert to Forth, I enjoyed Timothy Huang's article in the June 1984 issue of *DDJ* (No. 92). I agree that Forth should translate very well into Chinese. However, the use of the old phonetic symbols to spell out Chinese words is a giant step backwards. Overseas Chinese may know the old phonetics, but I believe that these symbols are little used on the mainland now.

It would be quite straightforward and much better to use China's own *Pinyin* for this purpose. This is not just another phonetic system; rather the Latin alphabet has actually been incorporated into the Chinese language. The letters are used for teaching and to provide a precise description of speech sounds.

For example, the Forth word OR could be typed in as HUOZHE; the Chinese will see this as the normal word for "or" (the regular tone marks *could* be used: HUO`ZHE´, but programmers could probably do without them and simplify the code). Likewise, ROT would be FANZHUAN, DROP becomes DIUDIAO, and DUP is CHONGFU. The byte-rich *Pinyin* versions could be shortened, e.g., like our ROT and DUP.

The main advantages would be that the words would be real Chinese words, and foreign computer consultants (who could already know *Pinyin*) would have easy access to the code.

Huang refers to the significance of the billion Chinese; most of them are on the mainland, however. Sooner or later, the overseas Chinese will have to come to terms with the big changes that have been made in the Chinese language during the last generation. In writing Forth, these changes may actually benefit the programmer.

Roger V. Swearingen
5333 Baltimore Dr., #158
La Mesa, CA 92041
(619) 697-3290

And the Shotgun Approach

Dear Doctor,
I must say that, as a new subscriber, it is a pleasure to have an issue arrive that demonstrates my own good judgment with regard to magazine dollars. While I'm not a C programmer, the C articles were a font of interesting algorithms and ideas. The core of my interest, though, was in the three articles directly addressing my interests—CP/M on the Commodore 64, Chinese Forth, and decision and cognition theory in Comments on Sixth Generation Computers.

With regard to CP/M on the C-64, I would add to the author's description of how to patch around having two 1541 disk drives the admonition that the 1541 was not really built for this job in the first place, and if you are going to have more than one disk drive on the Commodore-64, you should seriously consider getting a dual drive unit in the first place. The second comment is that, beyond the drive format availability, there is the fact that the 64 has a 40-column screen, and while you can get around this, it comes at a price that belies the 64's original value. Yes, there are a lot of Commodore 64s out there, but no, there are not a lot with CP/M; this situation is stuck in the old loop that it will take 64-formatted disks available in the real world to make CP/M a more popular option for the 64, and it will take more 64s with CP/M to make more 64-format disks available. The screen makes it all that much less likely.

Timothy Huang's article on Chinese Forth was very intriguing. It would be nice to see what matrix representations of the listed characters would look like and what resolution is necessary to produce them. I might also comment that in a subroutine-threaded Forth (in 6502), C: could simply be

```
: C: create -4 allot JUMP C; ' , ;
```

that is, back up over the 4-byte subroutine-threaded create routine, put the constant for the processor's jump instruction ($4C for the 6502) into the dictionary, and follow it with the cfa of the primary instruction. This demon-

strates the versatility of subroutine-threaded Forth—with direct executable machine instructions, you can jump directly into a definition. (For a discussion of subroutine-threaded Forth, see the June *Byte*.)

Ah yes, but the almost-gem of the issue was Michael Doherty's continuation of a dialogue on the future of computing. Almost everything Mr. Doherty said is towards the point, if not quite to the point. With a slightly stronger base in physiology, he could have come right out and said it—the human brain is not, repeat, not a Turing machine. We don't know exactly what it is yet, but we do know that the basic unit of the human brain is not only switched but also magnitude modulated, and that its output may act so as to switch other neurons, to modify the magnitude of their responses, or to do both simultaneously with the same or distinct target neurons. Push or pull it any way you prefer, and the result is still that the basic building block makes a difference. It may in fact *not* be possible to build a Turing machine with magnitude-modulated components; it is at any rate clear that this did not occur with the human brain. Here perhaps the useful metaphoric distinction is not that between quantity and quality but between number and quantity. You always have a distinct number of things. You never have a distinct quantity of something. When we improve our measurement capability to measure, say, an exact quart of liquid, we turn and look and, lo, what we have found is the number of atoms in a quart, and we have turned from measuring quantity to number. Please note that I am using these words in their squishy English senses and not in the precise Latin quantum of physics.

So much of Mr. Doherty's commentary was on the ball. He slips up on the discussion of information storage, however. He confuses information with the media on which it is stored. His oversight is not an uncommon one, after all. Who has seen information that isn't stored in some medium or other? Good question indeed, and the answer may just be everyone, for we don't know and are not yet even ready to address the question of where the quantum information of a quantum particle *is* exactly.

The reason that this is to the point is that, if this quantum information is in a null space, then it would be, in actuality, medium-free information, because storage media don't fit in dimensionless spaces. Well, if quantum information can be there, it is possible to entertain the notion of keeping other information there, and then the storage "space" constraint on information is quite meaningless. That doesn't affect the main question of what will be processing the information, and it's not too hard to see that the obvious next step after putting discrete processors in parallel is to put them in parallel with distributed analog processors. This leaves open whether we can engineer better analog processors than the von Neuman machines running around putting out provocative magazines and the like.

Yours,
Bruce R. McFarling
600 W. Broadway #6
Granville, OH 43023

[*More letters on Sixth Generation dialogue next month. — Ed.*]

Turbo Tidbits

Dear *Dr. Dobb's*:
I enjoyed the review of Turbo Pascal in the June issue. I have been using Turbo Pascal for several months, and it is an excellent program, fully deserving the praise in the *DDJ* review. I thought it would be useful if I listed some of the problems I have discovered in the Turbo Pascal system. I am presently using version 2.0 of Turbo Pascal on an 8-bit (CP/M) system. As far as I have been able to determine, virtually all of the following persist in both versions 1 and 2.

1. The installation program reverses the definition of screen hiliting and unhiliting commands. There are several sources of confusion in the terminal installation program, which manifest themselves when nonstandard installation is attempted.

2. The listing program, TLIST, has several bugs, the most serious of which result in missing characters or lines and an improper number of lines per page, which puts the page break in the middle of the page for long programs.

3. In the 8-bit version, recursion is not permitted unless a special compiler directive is used. However, no error message appears either at compilation or run-time when code is recursive. This is a serious problem, because recursion may be indirect and hence not obvious (procedure A calls procedure B which calls procedure C which calls . . . procedure A).

4. When inline code is used, a variable identifier may be used, which, according to the manual, is replaced by two bytes which contain the memory address of the variable. In fact, the variable name is sometimes replaced by the address of the variable and sometimes by an address which contains the address of the variable (indirect addressing). Which of these is the case depends on whether the variable is global, local, a var parameter, or other parameter, and in some cases how the variable was used previously in the program block.

5. Turbo Pascal programs destroy all but about the first 30 characters in the command line buffer.

Even with these problems, Turbo Pascal is a great program and a bargain. However, since Borland was notified of some of these problems well in advance of the version 2.0 release, it is unfortunate that they didn't make any corrections in the new version or even include a list of known bugs in the documentation.

Sincerely yours,
Harry Demarest
Astro Research, Inc.
Box 74
Corvallis, OR 97339

Dear *DDJ*,
Regarding your review of Turbo Pascal in the June 84 issue, I agree that generally Turbo is a vast improvement over other compilers. There is one problem with Turbo which I find very disturbing, however, and that is the way it handles (?) integer overflow. The program (see figure below) illustrates the problem. A total program crash would be much more acceptable than to have this sort of viper lurking around. To Borland's credit, section 3.1 of their manual plainly states what will occur, and there is even an entry in the index, but still how hard could it be to fix this?

Donald G. Simpson

1121 Manchester Dr.
Raleigh, NC 27609

Dear Sir:

In the review of Turbo Pascal in your June issue, there is an apparent misunderstanding of the **with** statement. I refer to material in the righthand column of page 76. The following Pascal excerpts should clarify the matter.

```
    type
        rec2 =  record
                    s1 : integer;
                    s2 : boolean
                    end;
        rec1 =  record
                    r1 : integer;
                    r2 : rec1
                    end;
var
    a : rec1;
begin
    with a.r2 do
      s1 := 5;
    writeln(a.r2.s1)
end
```

This will obviously output the integer "5". Now amend the code part to read:

```
    begin
        with a do with r2 do begin
          s1 := 5;
          r1 := 4
        end;
        writeln(a.r2.s1,' ',a.r1)
    end
```

The output will be "5 4". Obviously the nested **with** statement qualifies with "a" where applicable and with "a.r2" where applicable. Finally, "with a,b,c . . ." is an abbreviation for "with a do with b do with c do . . .". So, the above fragment can be written:

```
    begin
        with a,r2 do begin
          s1 := 5;
          r1 := 4
        end
        writeln(a.r2.s1,' ',a.r1)
    end
```

I hope this sheds more light than it does confusion.

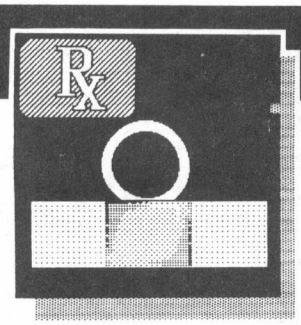
Yours very truly,
Herbert Kanner
211 Washington Ave.
Palo Alto, CA 94301

DDJ

```
      Line 1 Col 1 Insert Indent B:INTBUG.PAS
program IntegerBug (input, output);
{ Demo to display results of integer overflow }
{ Turbo Pascal v2.0 MS-DOS configured for Zenith Z-100 }

var n: integer;

begin
      ClrScr;
      write ('Enter an integer between −32,767 and +32,767: —> ' );
      readln (n);
      writeln;
      writeln ('The integer you have entered is: —> ', n);
      writeln;
      writeln ('Twice the integer you have entered is: —> ', 2 * n)
end.

Enter an integer between −32,767 and +32,767: —> 15000
The integer you have entered is: —> 15000
Twice the integer you have entered is: —> 30000
>
Enter an integer between −32,767 and +32,767: —> 32767
The integer you have entered is: —> 32767
Twice the integer you have entered is: —> −2
>
```

Figure

DR. DOBB'S CLINIC



by D. E. Cortesi, Resident Intern

An Optimizing Technique

Early this year, your intern gave vent to the frustration he feels at the primitive designs of compilers for personal computers. All (or, at any rate, most) of them are simple one-pass, recursive-descent parsers. If you don't know what that means, study the Small-C compiler; it's a textbook example. In a recursive-descent compiler, the flow of control in the compiler echoes the syntactic structure of the source program (the syntax graph of the program at any point is described by the state of the compiler's dynamic call-stack when it has analyzed to that point). In a one-pass compiler, the object code for each text element is emitted as soon as that element has been parsed.

Such compilers are (relatively) easy to design, and they run quickly. However, they are incapable of doing any kind of global optimization—this operation requires the compiler to make multiple passes over the program. In a multipass compiler, the first pass (*which* can be designed as a recursive descent) validates the program's syntax, collects information in the symbol table, and probably builds a tokenized version of the expression text for later processing. Later passes can apply the information found in the first pass in such a way as to make the final object code smaller, faster, or both.

Several readers responded with long, thoughtful letters on the place of C and its compilers. David S. Tilton, for example, made the point that because C is a low-level language designed to give the programmer close control of the hardware, we shouldn't expect a C compiler to do much optimization. He said, first, that the language is so simple there aren't many opportunities for a compiler to do optimization, and second, that the language gives programmers the tools they need to do the optimization themselves.

Another respondent, Mike Meyer, made an architectural point. He pointed out that most data references in a recursive language such as C are to local variables and hence to the stack. "Unfortunately, the 8080 family just doesn't support such things well. What a local reference turns into is 'get the item so many words from the frame pointer.' On an 8080, this translates into several instructions just to load a local."

Well, these letters got us thinking. In 8-bit machines, as Meyer says, the code for accessing a variable with the dynamic storage class is much slower than that for accessing a variable with the static storage class. Local variables of a function are dynamic by default, but the programmer can give them the static storage class.

Now, every book on C programming advocates the use of lots of small functions—each with its handful of local variables—and these locals are just the pointers and counters that are most heavily used. But if you are using an 8-bit machine, all those locals will cost you heavily. "Everybody" knows that for best performance at least some of them should be declared static.

There are problems in doing so. You have to decide which functions are in the critical 20% and which of their locals are really important; you have to modify the source text and recompile. These are clerical tasks that the computer should do. Furthermore, if you make a local variable static in a recursive function, you'll cause a tricky bug. It isn't always easy to tell which functions can be called recursively and which cannot, especially if you are modifying someone else's program.

Finally, there is no speed penalty on dynamic locals in most 16-bit machines. In the 8086, for example, it costs no more time to access the stack frame than it does to access the data segment. Yet, if your 8-bit program is ported to a 16-bit machine, those static local declarations will port right along with it.

This seems like a perfect example with which to refute Tilton's point—a case in which the programmer could *not* do a decent job of optimizing and in which the compiler, even an 8-bit compiler, *could*. Shouldn't an 8-bit C compiler be able to tell which local variables should be static and which not? At the very least, the compiler could tell which functions contain no function calls at all; their locals certainly can be static. Furthermore, no two such functions can ever be active at the same time, so the locals of all such functions could be allocated in the same area of static storage.

Well, it's all very well to say a compiler *should* do these things, but is it really a practical thing to ask? We sat down one day and worked out the optimization scheme that follows. After describing the method at a high level, we'll consider its space and time requirements.

The Well-Behaved Function

The input to a C compiler is a source text that declares one or more functions. Each function is public; it can be called from other compilations.

The functions contain expressions, and an expression can include a reference to other functions. A function that refers to itself is directly recursive. A function can be indirectly recursive, in that it calls a function (that calls a function, and so on) that calls the first function again.

A reference to a function that is not declared in the same text is an external reference; the compiler can't know anything about the called function. In particular, a function that contains an external reference is potentially recursive, because the external function

might call (a function that calls, and so on) the first function again.

The functions we want to identify are those that are not recursive. Because the compiler's information is limited, we will take a conservative approach, assuming a function is recursive if there is the slightest possibility that it might ever be so. The functions we can identify as definitely nonrecursive we will call "well-behaved functions," or WBFs. A WBF can have its local variables allocated in static storage. We can give a recursive definition of a WBF:

- A function that calls no other function is well behaved.
- A function that calls only WBFs is well behaved.

The optimization scheme is based on this definition, but a lot of details have to be tended to first.

GLFs and TFs

The functions of the C standard library are named as external functions, yet they are almost always well behaved. We can think of only one case in which a library function is ill behaved. In some implementations, a file access function may, on encountering a disk error, make an indirect call to a user-written error function. In this case, the error function might conceivably call the function that called the library function in the first place, thus producing a recursion.

In the main, functions from the standard library are well behaved, and they are named in many user functions that, but for these references, would be seen as well behaved. Let us define a set GLF of good library functions that are always well behaved and use it to expand our definition of a WBF:

- A function that calls no other function is a Terminal Function (TF).
- A function that calls only GLFs is also a TF.
- A TF is a WBF.
- A function that calls only WBFs is a WBF.

We will detect WBFs in two stages: first, we will find all the TFs; then we will find the WBFs that are not TFs.

The Symbol Table

We are supposing a multipass compiler. The first pass performs the syntactic parse of the source text and, for every identifier named in it, builds a symbol table entry (STE). We are concerned only with the STEs built for function identifiers. Besides the other information the compiler needs, we will require these items in a function STE:

- glf: Boolean; true for a good library function
- idf: Boolean; true for an internally defined function (one defined in this source text)
- prf: Boolean; true for a possibly recursive function
- caf: Boolean; true when this function calls another function
- pwb: Boolean; true for a possibly well-behaved function
- twb: Boolean; temporary flag for well-behaved functions
- wbf: Boolean; true for a well-behaved function
- asl: Boolean; the aggregate size of the local variables declared in this function
- osl: integer; the offset of static locals (when used)
- map: integer; used in mapping during pass 2

The use of these items will become clear as we proceed.

Before Pass 1

Before the first pass begins, the compiler must prepare a list of names of GLFs. The list can be coded into the compiler or it can be read from an external file. The latter course would allow users to extend the list with names of their own GLFs or to remove names if they should modify a library function in such a way as to make it ill behaved.

During Pass 1

During the first pass, the compiler must collect information that will be used to identify WBFs. These actions are in addition to the other things it does while parsing the source text.

When an STE is created for a function, the added variables listed above must be initialized. Except for glf, all the Booleans can be initialized to *false*. The name of the function must be looked up in the list of GLF names and the glf Boolean set *true* if it is found and *false* otherwise. The contents of the integer items can be undefined. When a function identifier is parsed in an expression or an initializer, an STE will be created as described (or merely found, if the function was declared previously).

When a function declaration is parsed, its STE will be created (or found, if the function is declared after it has been referenced). At that time, the idf Boolean must be left *false* if the declaration has the external attribute or set *true* if it does not. After a nonexternal function has been parsed, the asl integer must be set to the aggregate size of the local variables declared in the function.

When a function call is parsed, the compiler must record it as a tuple (*source, target*) in a list of such call tuples. In the tuple, *source* is the address of the STE of the current function, *target* that of the called function.

If the call is indirect through a pointer, the called function is unknown. In that case, the tuple should be recorded as (*source, source*). This makes the call appear to be a direct recursion. That's reasonable, because an indirect call could conceivably represent a recursion, either direct or indirect.

At the end of the first pass, the list of GLF names has served its purpose. Its space can be reclaimed.

Identify TFs

When pass 1 is over, an STE exists for every function identifier, and a call tuple has been recorded for every call. Now we can identify the terminal functions (TFs). To do so, scan the list of call tuples and, for every (*source, target*),

- if (*source* = = *target*), set *source*.prf = *true*
- else if (not (*target*.idf or *target*.glf)), set *source*.prt = *true*
- else if (not *target*.glf), set *source*.caf = *true*

The TFs are functions that call no functions or call only GLFs. They can now be marked as WBFs. Because they can't be recursive, their locals can be static. Because they call no user-defined functions, no two of them can ever be active simultaneously; therefore, their static locals can be allocated in a space that is common to all of them.

We will need two variables to handle storage allocation: slbot will contain the base offset of an area for static locals, and sltop will track the high-

water mark of the area. Set both to zero. Then scan the symbol table and, for every function entry in which (idf and not (prf or caf)) is true:

- set wbf = *true*
- set osl = slbot
- set sltop = max(sltop,asl)

When code is generated for WBF, its locals will be allocated at an offset of osl in an area of static storage. The locals of a function not marked as a WBF will be given dynamic allocation as usual.

If it should prove that there are no TFs, nothing further can be done.

Map PWB Functions

We have established the TFs (which are also WBFs), but there are more WBFs to be found. They are a subset of the set of possibly well-behaved functions (PWBs). To locate this subset (actually multiple subsets), we need to establish a mapping from the PWBs to the integers 1 . . *P*, where *P* is the count of PWBs.

Allocate space for an array of pointers to function STEs and call it *F*. Then scan the symbol table for PWBs. A PWB is a function STE for which (idf and not(prf)) is true. This includes the TFs plus all the functions that contain neither an indirect call, an external call, nor a direct recursion. (Some of these may yet be recursive. We will never identify them as such, but we won't call them WBFs, either.) Initialize *P* to zero. For each function STE,

- set pwb = idf and not(prf)
- if not(pwb), then iterate, else
- increment *P*
- set map = *P*
- set *F*[*P*] to the address of the STE

Now the functions *F*[1 . . *P*] are the (possibly) well-behaved ones, and the array *F* serves to map them to the integers 1 . . *P*. If there are none, of course, nothing further can be done.

Cross-Map Function Calls

We must prepare an array of Booleans *B* having *P* rows and columns. Each row *B*[*r*,*] will represent calling function *F*[*r*]; each column *B*[*,*c*] will represent called function *F*[*c*]. Allocate the array and clear its elements to *false*.

Fill in the array with a scan of the call tuples. For each tuple (*source, target*), if both *source* and *target* are PWBs, set *B*[*source*.map,*target*.map] to *true*.

After this is done, the list of call tuples has served its purpose; its space can be reclaimed.

Discover WBFs

We will now make multiple passes over the array *B*. On the first pass we will find those functions that call only TFs. Because they do, they are WBFs; hence, their locals can be static. Because they don't call each other, no two of them can be active at once; hence, their locals can be allocated in a pool that is common to them (but distinct from the space used by TFs). In each later pass we will find a set of functions that call only WBFs found in prior passes. The same comments apply to each set of functions.

To be sure that the functions found in each pass don't call each other, we will use the twb flag as a temporary note of a function's well-behavedness until the pass is complete. We will also count the number of WBFs we discover. When we make a pass that discovers none, we will stop. The logic of the outer loop is:

- set discoveries = 0
- set slbot = sltop
- for *r* from 1 to *P*, perform inner procedure
- if discoveries = 0, break
- for *r* from 1 to *P*, if *F*[*r*].twb, mark *F*[*r*] a WBF

To mark a function as a WBF, we do these things:

- set twb = *false*
- set wfb = *true*
- set osl = slbot
- set sltop = max(sltop, slbot+asl)

The inner procedure inspects one function to see if it is well behaved. It needs to be performed only for functions that are not yet known to be WBFs. Its purpose is to set the *F*[*r*].twb flag.

- if (*F*[*r*].wbf), return
- set *F*[*r*].twb = *true* and perform inner loop

The inner loop runs a variable *c* from 1 to *P*. For every *B*[*r*,*c*] that is true, it checks *F*[*c*].wbf; if it is false, then *F*[*r*].twb must be set *false* and the inner loop can end.

At the completion of this operation, the WBFs have been identified in non-conflicting groups. Each group has been assigned an offset for its static locals, and sltop contains the aggregate size of all static locals.

Performance

Through the point of identifying the TFs, this method shouldn't impose much of a load on a compiler, even an 8-bit compiler. The code added to pass 1 and the code to find the TFs are fairly small. The primary cost is in three blocks of space: the list of GLFs, the list of call tuples, and the extra items in a STE.

If space is a problem, the list of names of GLFs need not be complete. The list might contain only the names of the most common library functions. The more names in it, the more TFs that can be found. However, the only effect of omitting some GLFs is to make PRFs of some functions that could have been TFs; these will have dynamic locals when they could have had static ones, but the object program will still run.

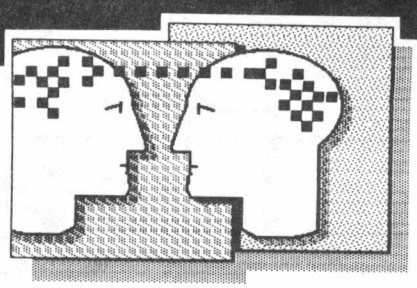
The list of call tuples can also be truncated. It could be allocated in a small area of fixed size. Each tuple needs only 4 bytes, so an area as small as 256 bytes could contain up to 64 tuples—a useful number for small programs. When a function call is parsed and the tuple list is found to be full, pass 1 would mark the source function as a PRF and not record a tuple. As a result, a function whose calls were not completely recorded as tuples could never be identified as a TF or a PWB.

The extra STE information could impose the largest cost, because it adds several bytes to every symbol-table entry. However, that need not be true. Not every STE describes a function. The information unique to a function could be allocated separately and dynamically when a function is entered in the symbol table. The basic STE format would only need to allow room for a pointer to the function information, not the information itself. Furthermore, some of the STE items used here can be overlapped. The map integer may be a union with the osl integer, and the pwb and caf flags can be a union.

The cost of the second phase, locating other WBFs, is harder to analyze. It requires *F*, a vector of addresses of length *P*, but *P* is not likely to exceed

(Continued on page 21)

CP/M EXCHANGE



by Robert Blum

Everybody apparently has at least one true-life story to relate when the topic of conversation turns to humiliation. I'm no exception. I used to have several favorite stories; I now have another.

Several months ago I decided that I soon would have to replace some of my then dreadfully old S-100 hardware because I was beginning to experience excessive down time. Except for the cost of an upgrade, I wasn't at all upset at the prospect of buying some new hardware. I had been waiting quite a while for a sound reason to move ahead and bring up a higher performance system.

Many new high performance boards had become available since my original purchases. This caused me great consternation when deciding what I should buy. I wasn't unhappy with the performance or features of the boards I had at the time; they were simply old and needed replacement. As it finally worked out, I didn't find anything that better served my needs.

I elected first to renew my 4 MHz CompuPro Z80 CPU board with a more recent revision level of the same model, capable of 6 MHz operation. I knew my present memory cards would function perfectly at the increased clock rate because they were rated at 12 MHz, as were the I/O boards I was using.

I was not nearly as confident about the floppy disk controller that I had purchased from a different manufacturer. Of all the boards in my system at that time, the floppy disk controller was the most complex and the one that I knew the least about. I wasn't sure that I would be able to find a hardware solution to a speed-related problem if one were to occur. However, if at all possible, I wanted to keep the board because of the time I had spent fine-tuning my BIOS code. The thought of reinventing the wheel for another disk controller board didn't excite me in the least. Fortunately, a phone call to the manufacturer brought good news: I could upgrade to the latest revision level board, which was guaranteed to be 100% software compatible with the old one and to run flawlessly at 6 MHz, all for only a nominal upgrade charge.

While waiting for the new disk controller to arrive, I ran diagnostics constantly for several days. During that time not one error occurred that I could attribute to the new hardware.

Soon the new disk controller arrived. I wasted no time getting it ready for use and couldn't have been happier when it booted for the first time from drive A:. All seemed rosy until several programs on the A: drive turned up lost. Listing the directory showed that the programs were definitely missing; in their place were several meaningless filenames. I began to feel a bit uneasy because the last time I saw a display of this type was when I helped a friend rebuild his disk directory after a mysterious system crash.

I executed my disk utility program, which allows sectors to be displayed in hexadecimal, to see what was causing these unknown filenames to appear. I started my search with data group zero, which on all standard CP/M systems is the start of the directory area. About halfway through data group one, I found a sector that had been completely zeroed. I didn't have a hint as to what could have done this but suspected that the disk controller might be more clock rate sensitive than I had been led to believe. I slowed the CPU down to 3 MHz and inserted another disk into drive A:. Again the system booted on the first try, but this time it ran without destroying the directory.

This verified my first suspicion that the disk controller was speed sensitive. However, the manufacturer had told me that running at a faster rate would be no problem. I called again to find out if I had done something wrong or possibly had not configured the controller board correctly. I was told to check several wiring changes and was again assured that all would work as planned after the bugs were shaken out. I dutifully went about checking for any mistakes I had made during setup—without result.

Having thoroughly checked the hardware, I turned to my BIOS. The only area I could find that appeared in the least suspicious was a section of code in the warm boot routine that flushed any active deblocking buffers to disk when a program ended. I thought there was an outside chance that the disk heads were being moved before the deblocking buffers were written. I couldn't imagine this happening but commented out several instructions for testing purposes. After installing the modified version of the BIOS on the system tracks of a test disk, I again speeded up the CPU to 6 MHz and booted from disk. Once again, after several disk accesses, a sector in the directory was destroyed.

At this point I had tried everything I could think of. Rather than continue to shoot in the dark, I turned my troubled machine over to a hardware magician for repair. Several days later my system returned running flawlessly at 6 MHz. I was told that the solution was a few minor hardware adjustments.

For months now I have been running the system without any further hardware problems. However, a few software situations have cropped up. I use a fairly well-known data base package for most of my application-oriented tasks. After installing the latest release of this software, I suddenly began losing large portions of my indexed data files. Investigation showed that the data content of all the records was present in the body of the file but the indexes were being corrupted. Several phone conversations with the technical support people

proved fruitless, so I wrote a couple of test jobs that I knew would consistently fail on my system and sent them along for analysis with my distribution disks and a *problem report*.

A short while later the support people returned my package with a note saying that my test jobs had failed on their machine as well. They further acknowledged that a problem exists but they were unable to pinpoint the source since it appeared to be a CP/M idiosyncracy. In the meantime I could apply a temporary fix, which involved adding several instructions to each of my jobs to ensure that the data buffers were saved to disk after each update.

A practically identical situation reared its head a couple of weeks ago. I was in the process of installing a new release of my favorite BASIC interpreter. After I had completed the configuration process and was trying to execute the new interpreter, a message appeared on my CRT instructing me to run the configuration process before using the BASIC runtime system. That sounded like a reasonable request—except I had already completed that step.

A call to the software manufacturer proved to be a valuable lesson for me. He informed me almost right from the beginning of our conversation that my problem was due to a faulty BIOS. He explained that his configuration utility reads the first sector of the program and modifies it according to answers given by the operator. At the conclusion of this interaction the changed sector is rewritten to disk and the file is closed. Unfortunately, CP/M 2.2 does not automatically flush deblocking buffers when a file is closed. Under normal circumstances this procedure works fine; problems occur if the BIOS warm start routine does not flush the remaining deblocking buffers when a program ends. This wasn't exactly what I wanted to hear, and I held my ground by refusing to give any credence to a possible bug in my BIOS.

The fellow with whom I was talking wasn't about to be swayed from his convictions, which made me think twice about what he had to say. It didn't occur to me until the next day that months ago I had commented out several instructions in the warm start routine of my BIOS while trying to find a disk controller problem and in my haste had not returned them to their original order. After I did so, both of my software packages began to run properly.

If you have experienced problems of this type, you might want to look through the warm start portion of your BIOS listing for a sequence of instructions similar to the following:

```
LD    A,HSTWRT      ;IS HOST BUFF
                    ;ER ACTIVE
OR    A             ;*
CALL  NZ,WRITEHST   ;YES—WRITE IT
                    ;TO DISK
```

Depending on whether your BIOS was written from scratch or patterned after the DRI examples, the instructions in the listing may not be even close to those in your BIOS. Nonetheless, what you are looking for is a section of logic that tests for and flushes any remaining active deblocking buffers. **DDJ**

(Continued from page 18)

100, so F is unlikely to occupy more than 256 bytes. Furthermore, F could be built in the space previously occupied by the list of GLFs.

The array B has $P*P$ elements, so it could become quite large if it used one byte per element—10K bytes when $P=100$. If it were stored as one bit per element, however, its size would be reasonable. In any event, the arrays F and B are transient; they can be discarded as soon as this part of the compile is finished.

The time cost of the second phase is potentially large. Each pass through the outer loop takes time proportional to P; there might be as many as P passes made. The inner loop takes time proportional to P, so at first blush the running time appears to be proportional to P cubed. It is not quite that bad, however, because the inner loop is only applied to functions that are not yet marked as WBF. Initially all the TFs are so marked, and more functions are marked on each pass. Furthermore, the inner loop can break off as soon as it is known that the current function calls a function that is not known as a WBF.

In fact, the inner loop need not be a loop at all, if B is stored one bit per element. The wbf flags of the functions $F[1\,..\,P]$ could be collected into a bit vector before the outer loop commences—while building F, for example. Then the inner loop could be reduced to anding the current row of B with that bit vector and comparing the result to the current row of B. This effectively removes the loop from the inner procedure.

That done, the time cost of the second phase becomes proportional to $2P$ times the number of passes made. The number of passes will be roughly proportional to the depth of nesting of function calls in the compilation. At one extreme, the compilation might define a chain of functions, A calling B calling . . . calling Z, a TF. Then one WBF would be found on each pass, and P passes would be made. At the other extreme, the compilation might define a set of independent TFs (and PRFs, but those are irrelevant), and only one pass would be made that would find no more WBFs.

Summary

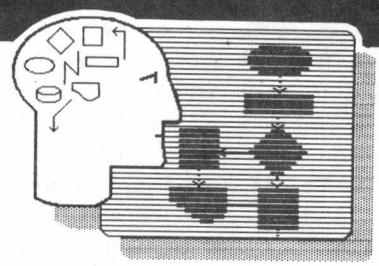
In summary, we've outlined a method by which a C compiler could locate sets of functions whose local variables can be static and allocated in common pools. The time and space costs of the method are small for finding the first set (the TFs) and probably reasonable for finding many sets.

The method is described in the context of C, but it ought to be applicable to Pascal or Modula-2. Furthermore, in languages where every function is not automatically public, the TFs and WBFs have other useful properties. TFs are potential candidates for in-line expansion as macros, for example, and all WBFs are candidates for use of a short-circuit protocol for procedure calls that doesn't require a full stack frame.

We'd be delighted to hear from any designer who actually tries the method out or from any reader who finds a flaw in it.

DDJ

THE SOFTWARE DESIGNER



by Michael Swaine

The following discussion never took place.

The software designers and critics whose words appear in this article have never gathered as a group to argue the design of software. If they had, it is doubtful that the precise exchange of views pieced together here would have resulted. But all the words I have quoted were actually spoken by the persons to whom they are attributed, and I have attempted to avoid changing the sense of any utterance in changing its context. I don't pretend that I have left every nuance intact.

The error of interpretation I am most likely to have committed is to make the speakers appear more dogmatic or polarized than they actually are. I have placed opposing views next to one another here to focus attention on the differing views; the undesired side effect of this is to make the contrasted views read more strongly than they would in isolation (or in their original context). Just remember that these people are all more complex individuals than their two-dimensional projections onto these pages.

The Participants

Rob Barnaby is the author of one of the best-selling software packages in history, WordStar. The compleat programmer, Barnaby delivered the original WordStar to MicroPro on disk, debugged and ready for duplication, along with camera-ready copy for the manual.

Lee Felsenstein is a hardware designer whose credits include the Pennywhistle modem, the Sol-20, and the Osborne 1.

Paul Heckel is the president of Quickview Systems, a software designer, and the author of *The Art of Friendly Software Design.*

Philippe Kahn is the president of Borland International, maker of Turbo Pascal.

Giacomo Marini is the vice president for software development for Logitech, maker of Modula-2 compilers.

Anthony Skjellum is president of Pyramid Systems, a software designer, and *DDJ*'s C and Unix columnist.

Pierluigi Zappacosta is the president of Logitech and formerly developed software on early 8-bit microcomputers.

It began last spring. Borland International's Turbo Pascal was beginning to attract attention, and editor Reynold Wiggins and I drove to Scotts Valley to talk with Borland's president, Philippe Kahn. In the course of our discussion, the following exchange took place:

DDJ: Turbo Pascal is doing very well. Sidekick looks interesting. What's next? Will you do an implementation of C?

Kahn: C is a disease. When I see people writing spreadsheets in C, I think, "They're out of their minds." It was designed to write operating systems. Modula-2 is good for that [writing spreadsheets]. We'll do a C. We'll do a C because everyone wants a C. But in Europe C is seen as an American disease, and here people are trying to spread it.

Now, C is a programming language originally developed at Bell Labs for use in writing operating systems; it has recently become popular as a general-purpose programming language. But not, apparently, with Philippe Kahn. Struck by the vehemence of Kahn's reaction, I started talking with software designers and critics about what they thought the real differences were between C and Modula-2 from the perspective of software design, and the result was this pseudo-roundtable discussion.

Zappacosta: I'm surprised, in a way, that C was invented on the East Coast, because it would be the perfect language to have been invented on the West Coast.

Barnaby: I don't follow that. I think it's more a distinction of academic and scientific vs. free-flowing, open-ended design work.

Heckel: I think the point is not East Coast vs. West Coast, especially since there's so much movement between coasts, but rather the academic environment vs. the let's-get-the-job-done environment.

DDJ: Bell Labs is not academic?

Heckel: OK, you might argue that Bell Labs is really an academic environment, but I point out that the charter of that lab was not to develop a language like C. C was a side effect. They decided that they wanted a simple operating system, so they developed an early version of Unix; they decided they wanted a high-level language, so they developed C. None of these things were what they were supposed to be doing. They were designing for themselves, focusing on tools for their own immediate needs rather than on an academic concern with how pretty an algorithm would look on paper. Modula-2, on the other hand, was a follow-on of Pascal that came out of the academic community. And Pascal was designed for pedagogical purposes, for the way the world ought to be.

DDJ: I think Pierluigi was drawing on a more general West Coast stereotype, the free-wheeling, laid-back Californian. And whether or not the stereotype has any validity, I wonder if there isn't something to the idea that C is well designed—even if less deliberately designed than Modula—for a freer, less-regimented kind of programming. And if so, what does that mean?

Zappacosta: Yes. On the West Coast,

people say, I want to do whatever I *want*; I am my own boss. Why not? In the States you can change your last name, so why shouldn't you use C?

Skjellum: Well, freedom is an issue. But it can be a very practical matter. You sit down to write a program and you don't want to fool around with a lot of academic rules and constraints.

Heckel: Obviously the Bell Labs people tried to integrate as much of the knowledge they had gained regarding compiler construction and design as they could, but without getting in the programmer's way, still giving people the freedom they need. When you're designing real operating systems and real compilers, you try to give yourself freedom rather than a straitjacket.

DDJ: It sounds like getting the job done is also an issue. Paul and Tony have both characterized C as a get-it-done language, and Paul has contrasted it to the structured, modular style of Pascal and Modula. But Lee, you've been talking to people recently about your own get-it-done approach to design, and what you've been saying sounds structured, modular.

Felsenstein: I'm getting into computer programming a bit more. First of all, I can't avoid it. No hardware designer today can ignore software. Secondly, for our project for the Community Memory terminal we have to make a bit of clever firmware. I intend to do that in one day, but I don't intend to do it myself. I'll spend about three weeks working out the structure of the system without getting down to the details. Those will then be filled in by a small army of coders—volunteers, in this case—who will have received specifications and will sit down for a few hours to code small routines.

DDJ: You're approaching software design from the point of a hardware designer.

Felsenstein: It's similar to building at the level of the chip. You work from top down, build a structure, and at the last minute fill it in. To build software like hardware, I am finding, requires a fair amount of discipline to avoid rushing ahead, and it also requires that you be able to visualize the construct.

Heckel: *I'm* not against structured programming, but you don't hire a musician because he can read music better than the other guy; you hire him because he plays better than the other guy. The aesthetics of the algorithm is one of several factors one uses in evaluating software. The problem with the direction of structured programming is that there are more and more ways to do something wrong.

DDJ: But structured programming, modular design, the approach that Lee just described and that Paul talks about in his book, that's all language independent, right? C has perfectly good tools for writing structured programs, and you can write modular code in any language. So how, specifically, is C any "freer" than other languages?

Skjellum: C doesn't force as many assumptions on the programmer. Think about the assumptions that are forced on you, in many languages, by the compiler. For example, you are forced to use intrinsic functions. In Fortran, all the I/O is intrinsic, and the compiler makes assumptions to help that along. And there are some traps that this leads you into. Think of the Write statement; the expression within the parentheses can be quite complex, and you might expect it to be handled by the compiler, but the compiler just passes it along to the runtime package. In C, you would know better than to make that mistake; the compiler doesn't play favorites.

DDJ: In C, the compiler has no prejudice?

Skjellum: No; it neither helps nor hurts. In Fortran, Write statements are helped out by the compiler. Pascal is like that; it makes certain assumptions. Think about the Writeln command: you give it an integer argument and an object to print out. That rigidity is frustrating when you're trying to get a job done. What I want is to be able to define, say, my own I/O functions at a level where they look like the built-in elements of the language. In C, I can.

DDJ: Modula-2 is being touted as the language that Pascal should have been, but aside from its roots in Pascal and an impression that it takes some freedom away from the programmer, I think most of us are a little vague about what Modula does and doesn't do. Does it, for example, force the kinds of assumptions that Tony wants to avoid?

Marini: In our version of Modula, we include the source code for some things that can be customized. Pointer management, the loading of overlays, how you manage interrupts: you may want to customize these things. Also, we distribute the sources for three or four library modules that are in some way environmentally dependent: the keyboard module, the display module, the RS-232 module. You can take the supplied modules as examples for designing your own. The flexibility is limited, but it's there. But the thing that makes Modula the language of the 80s is that you have separate but not independent compilation. You have a check at compile time between modules, so if you import a procedure from another module, you have to write the definition for the exporting module. When you link, you are sure that all the modules are version consistent. You have strict consistency checks between modules.

DDJ: You must admit, that sounds constraining.

Marini: If you are the only programmer around, it can be a little *annoying. The* Modula approach certainly uses strong discipline. But as soon as you have three programmers working on a project, this is really invaluable. Modula can handle complex programming tasks. I think that this is critical. The only other language that does this sort of thing is Ada. For people who understand and like the advantages of compile-time checks and strong typing, Modula has the chief advantages of Ada without the complexity of Ada or the unavailability of reasonable Ada compilers.

DDJ: Can you quantify the distinction between simple and complex?

Marini: I think the big difference is between the one-programmer and the multi-programmer project. That is the first step. The second step probably happens at about four or five programmers. From there up to a thousand programmers probably doesn't change very much. The big step occurs when you go above four or five programmers. Then you get into the complexity of communication within a team of software developers that cannot be based on simply informal, verbal communication. You start needing, absolutely needing, written specifications. When you have four or five, one will leave within the year. The probability is that they will not all have the same

style, the same feelings, the same taste, the same choices. You need some sort of interface management. You may want a network, a common library, other little things that are not so little in the end because they consume a lot of effort.

Zappacosta: And the DoD certainly knows quite a bit about that, because they specified Ada.

Skjellum: Ada exists for people who are writing a program that has to run for 20 or 30 years and they have to be sure nothing will go wrong. They're so worried that something might go wrong in their program that they have to force excessive rigidity in the design modules. So they invented Ada. Ada has very specific rules and a tremendous instruction set. I think of it as the Language of Ten Thousand Features.

Zappacosta: In some sense you can say they went overboard; yes, maybe. But when you deal with the complexity of programming that they deal with, maybe you understand why they went overboard. I think that somehow Modula took the right ideas from the right places. It's not so complicated that you have to have the DoD pushing you to use it. But on the other hand, it's appropriate for complex development like the people who are reprogramming air traffic control software throughout the country. If you propose to these people to use C, they cannot. They have I don't know how many people working on the project, and they are not sure that things would work together. So C is not an alternative.

Skjellum: I don't know much about Modula. And I'd like to keep it that way. But I must say that there are some nice features of Ada: more flexibility in the defining of data types, for example.

DDJ: It seems that we all agree that C allows the programmer the freedom to do things in unorthodox ways, that Modula and Ada force the programmer into orthodoxy, and that the difference of opinion is over whether and when the loss of freedom is justified.

Zappacosta: The constraints of a language like Ada or Modula gain you something in functionality, but yes, the incremental gain costs something. There are two ways to go. One is toward simplicity—the pfs approach—everything is simple, manageable. With the other you get Ovation, Framework, more complicated pieces of software. Unfortunately, for certain applications you haven't a choice. If you want to control all the airplanes in the United States' sky . . . .

DDJ: You can't use the pfs approach?

Zappacosta: Probably not. Logitech is positioned in a strange place. We offer Modula-2 for the microcomputer, and in the microcomputer you have the choice you don't have on larger machines: simple or complicated. Certainly we are not going after the simple applications. The fact that after the 8086 version we came out with the VAX version shows that what we have in mind are serious software developers.

DDJ: Are you saying that you have to have a VAX to be a serious software developer?

Zappacosta: Well, even for applications that eventually go on a PC, development on a mainframe is becoming more common. That's why it's harder to start a software company. You have to have a VAX; if not a VAX, you have to have a 68000-based machine. When I started programming micros in 1977, we had a CP/M machine that was the target machine and the development machine. Back then, the software couldn't be too complicated; the hardware was most important. Now things are changing.

DDJ: Software is getting more complex, requiring team programming, which in turn requires the overhead management that languages like Modula, with consistency checking and strong typing and so on, offer?

Zappacosta: Yes, but besides the number of programmers, the other parameter to measure complexity of software is the need for this software to change and evolve. If you write software and never change it, if it is a great hack and nobody can put his hands in there, while it works you can accept that level of functionality and reliability and leave it there. But there are many applications that are alive. And then it's different. If you have to put your hands into a system continuously, then you have this problem of maintenance. You have to understand the code of another guy, you have to port the system to a new machine, and that's another discriminating factor.

Skjellum: But you can do that with C. You can write machine-specific software, but you can also write software *that is portable.*

Heckel: Yes, do you know the concept of *lint* in C? It's referred to in the C manual. The *lint* program takes a C program and it finds all the things that Pascal tends to find and it gives you a list of them. Then if you really want a program that doesn't have any of those kinds of things in it, you can fix the C program.

DDJ: There really are two philosophies here, aren't there?

Zappacosta: Well, you know, every time we present Modula we go the easy way. We say, if you plan to use Pascal, Modula is a better alternative. I think there is little discussion there: Modula is indeed the better language. But C . . . . Of course there is the large C market that we would like to tap, but we are careful. We are not eager to start a religious war. There are two different kinds of people who defend C. One says, I don't need another language. That's either because their requirements are not that tough and they really don't need another language, or because of ignorance. When Bill Gates told Bobo [Daniel Borel] that he didn't need another language, I think that that was ignorance. After all, considering that he made his living writing BASIC, it's not surprising that he sees already too much for what he needs. But he has a large company, and now even Microsoft is getting behind schedule and not completing things. Maybe it is ignorance and he is making a mistake not to consider alternatives. But other people like C as a matter of personal taste; it's an expression of personal freedom. This second group we take great care not to touch.

Skjellum: At least compared to Pascal, C is a more general-purpose language; that's why I like it.

DDJ: Even granting, for the sake of argument, that team programming is the wave of the future, still there are costs associated with every decision. What do you lose in moving from the approach of the lone programmer who codes down on the bare metal to the more rigid, Modula or Ada team programming approach?

Barnaby: It seems like to do that you've got to pretty well spec out what you're doing first. So you can divide it into pieces and design interfaces

among the pieces. Which means you have to understand what you're doing first. And that isn't how you create products that are leaps in the state of the art. You get a good idea, you divide it up very roughly into pieces, and later find that they weren't the right pieces and you redefine. I find that I have to be freewheeling. That's my experience in doing programming. WordStar grew out of a video program, and as I worked on it I kept coming up with a better way to do it.

DDJ: What one hears repeatedly is that many of the breakthrough software products are developed by programmers for themselves, not as products. Is that just folklore?

Barnaby: No. Look at Unix and C. I'm trying to get back to that approach after not having produced anything significant since WordStar and after having noticed that I wasn't approaching software designing in that way. I'm currently working with some very vague ideas that may lead to developing some new techniques to do certain things. It's creative exploration. It might or might not be incorporated into a product.

DDJ: So, Rob, you're going back to an earlier, freer approach to design. And clearly an individual approach. But Pierluigi and Giacomo, you seem to be saying that the times have outgrown that approach.

Marini: Three examples: VisiCorp, Microsoft with Windows,Digital Research with Concurrent DOS. In all three cases we have consistent out-of-schedule behavior. These were basically all companies that went from very small development groups to significantly larger development groups, and probably without project management.

Zappacosta: The fact that the microcomputer industry has been so innovative, so creative, probably comes from the fact that the past was forgotten. The industry was new; the machines were simple and small. And simple tools and smart people produced good results. They proved that there is also a side to programming that is not represented by the Mythical Man-month. But then the industry grew, the machine grew in power, and now to reach the vast majority of people you have to use a different interface. The traditional solutions used in the microcomputer industry are no longer adequate. In the Macintosh you have 64K of highly optimized, hand-coded software just to move the mouse. Unfortunately this trend may make creativity more difficult to foster.

Marini: Well, the market is getting more conservative. Interface compatibility is becoming a little like the story of the awful IBM keyboard.

DDJ: The awful IBM PC keyboard or the awful PCjr keyboard? IBM is backpedaling fast on the PCjr.

Marini: Maybe this is still reversible, but certainly the user interfaces to things like Lotus 1–2–3 are not easily reversible. The point is that the market builds up inertia.

DDJ: Lotus must be hoping that users can unlearn the 1–2–3 interface, or Symphony won't sell. But I guess you're saying that users now have expectations, and those expectations constrain what software designers can do with their creativity?

Marini: Yes. I suspect that the creativity is still there. Americans are so conservative. In the U.S. you can't change the color of the dollar. In Italy, we change the color of the money every three years.

DDJ

More dBASE II Programming Techniques

by Gene Head

In my last article (*DDJ*, June 1984) I described a machine language subroutine called ZIP-CHK.ASM that partially validated zip codes and could be used by dBASE II using the LOAD FILE, SET CALL TO ADDRESS, and CALL VARIABLE commands. I now realize that older versions of dBASE II do not support the LOAD FILE command that facilitates the loading of machine language modules from assembled HEX files. However, most early versions of dBASE II (pre v.2.4) *will* support the SET CALL TO ADDRESS and CALL VARIABLE, even though these commands may be undocumented in the user's manual.

In this article I will explain these and other undocumented features of early versions of dBASE II. I have access only to versions 2.3 and 2.4, so you should check other versions thoroughly to determine whether the feature actually works as I will describe it. I would appreciate hearing about any results you may have on versions prior to v.2.3; I will catalog these findings and pass them on in future articles.

"Most early versions of dBASE II will support the SET CALL TO ADDRESS and CALL VARIABLE. I will explain these and some other undocumented features of dBASE II."

In addition to the undocumented features, I have included a description of the HEX file format and a dBASE II command file that will simulate the LOAD FILE command if you can get the undocumented PEEK and POKE to work. After reviewing the format of the HEX file, you should be able to write a LOAD simulator in any high level langauge using LOAD-HEX.CMD (listing, page 27) as a guideline.

Gene Head, 2860 NW Skyline Dr., Corvallis, OR 97330.

Pre-v.2.4 Undocumented Commands

These undocumented features appear in early versions of dBASE II. PEEK (aaaaa) will return the contents of a decimal address; POKE aaaaa,nnn will fill a decimal address with a decimal value; SET CALL TO aaaaa will set the call address to a decimal value; and CALL VARIABLE will call the address as defined in the SET CALL TO command and pass the string VARIABLE, as described below.

Upon entry of the subroutine, the HL register pair points to the location of the passed variable. The variable is stored in the format LENGTH BYTE followed by the actual string characters. For example, if the variable ZIP represents the string 97330, then upon CALLing a subroutine HL will point to a series of locations with the following data bytes in hex: 05, 39, 37, 33, 33, 30. The HL pair points to the 05, indicating there are five bytes in this string; 39, 37, 33, 33, 30 are the hex bytes that represent the ASCII string 97330. You can modify the string, but you must *not* increase the length of the variable. If the string becomes shorter, you should fill it with trailing blanks.

TEST(variable) will test the variable and return the following: −6 if the value is NUMERIC type, 0 if the variable does not exist, 1 if the variable is LOGICAL type, and nn (a number be-

tween 1 and 255) to indicate the length of a STRING type variable.

Loading a HEX File

The HEX file is simply an ASCII text file that holds information about what bytes go where. The LOAD.COM program usually is used with the HEX file to create an executable COM file. The format of any HEX file is shown in the figure at right.

```
:10A470002356235E22F0A401080021F2A47EFE07E9     <— line 1
:10A48000CACFA4BACA8BA409C37DA4237EBBCA9633     <— line 2
          (30 lines were deleted in this example)
:10A67200575632343732363857493533035343914     <— line 33
:09A68200575938323038333100E9                   <— line 34
:0000000000                                     <— line 35
```

Figure

In a HEX file every line ends with a carriage return followed by a line feed, and every line begins with a colon. In line 1 (see figure) the two bytes following the colon (10) are two ASCII characters that represent the HEX value in the range 00 – FF (0 – 255 decimal). I will call this value the NUMBER OF BYTE-PAIRS value. We will get back to it later.

The next four bytes (A470) are ASCII characters that represent a HEX value in the range of 0000 – FFFF (00000 – 65,535 decimal). I will call this value the FIRST LOAD ADDRESS.

The next two bytes (00) are ASCII characters that are reserved. That means I don't know what they are for. However, in every HEX file I have ever looked at these two bytes are always 00, and for this article I don't think it is necessary to define them exactly. The same can be said for the last two bytes on each line (E9 in line 1). A checksum perhaps?

The intervening bytes (02, 35, 62, 35, etc.) should be viewed as byte-pairs. Remember the NUMBER OF BYTE-PAIRS mentioned earlier? Well, that is exactly the number of these byte-pairs we will load into consecutive memory starting at the FIRST LOAD ADDRESS. Each of these byte-pairs are ASCII characters that represent a HEX value in the range of 00 – FF (0 – 255 decimal). Given the example of line 1, the address locations should be filled with the corresponding byte values as follows:

A470	23	A478	08
A471	56	A479	00
A472	23	A47A	21
A473	5E	A47B	F2
A474	22	A47C	A4
A475	F0	A47D	7E
A476	A4	A47E	FE
A477	01	A47F	07

If your high-level language does not support a LOAD function, you can simulate one as long as you can read a textfile and PEEK at and POKE into memory. The command file LOADHEX.CMD (see the listing) was written for dBASE II. dBASE II only understands decimal numbers, and most of the code is for converting two ASCII bytes to a decimal number to be POKEd into memory. Understanding the format of the HEX file should enable you to write your own HEX loader simulator in any language.

Using LOAD-HEX.CMD as a guide, process line 1 of the HEX file. Extract the load address (position 4-7 in the string), the number of byte-pairs to POKE (position 2-3), and the individual byte-pairs (beginning at position 10). POKE each byte-pair then process the next line of the HEX file. Continue processing sequential *lines until the* NUMBER OF BYTE-PAIRS is equal to zero. The HEX file is now loaded into memory and ready to CALL as a machine language subroutine.

DDJ

dBASE II Listing

```
*       LOAD-HEX.CMD
*
*       File to simulate the LOAD feature of dBASE II 2.4 in
*       earlier versions that do not support this feature.
*
*       Verify that PEEK and POKE commands work before typing
*       in this file and watch the syntax especially '(' & ')'.
*
*       From:   Head Quarters
*               2860 NW Skyline Drive
*               Corvallis, Oregon  97330
*               (503)  758-0279
*
*       Copyright 1984 by Gene Head - All rights reserved
*       For private, non-commercial use only.
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
SET TALK OFF

* --->  First time initialization
*
* The following need be done only once
*
* First CREATE a file named HEXDATA.DBF with ONE
* field NAMED DATA of CHARACTER type, forty-four
* (44)  characters in length.
*
* APPEND FROM the SUBROUTINE HEX file using the SDF option.
* If you were going to use the Subroutine ZIP-CHK.HEX file then:
*
*       .
*       .USE HEXLOAD
*       .APPEND FROM ZIP-CHK.HEX SDF
*       .
*
* <---  End of first time initialization
*  Each line of HEX data will be in the field named DATA
USE HEXDATA

*   POSITION IN THIS STRING IS THE DECIMAL VALUE OF THE HEX CHARACTER
*    (SEARCHING FOR A '0' RETURNS 0)
STORE '123456789ABCDEF' TO HEX

* --->  COMPUTE THE BASE LOCATION OF THIS LOAD (THIS FUNCTION IS OPTIONAL)
STORE STR(((@($(DATA,4,1),HEX)*16 + @($(DATA,5,1),HEX)) * 256) +;
  (@($(DATA,6,1),HEX)*16 + @($(DATA,7,1),HEX)),5) TO CALL:ADR
SET CALL TO &CALL:ADR
* <---  END OF LOCATION OPTION

DO WHILE $(DATA,2,2) <> '00'
  * Compute how many bytes to POKE from this line of the HEX file
  STORE (@($(DATA,2,1),HEX)*16 + @($(DATA,3,1),HEX)) TO COUNT

        *  Get the starting POKE address
        STORE ((@($(DATA,4,1),HEX)*16 + @($(DATA,5,1),HEX)) * 256);
         + (@($(DATA,6,1),HEX)*16 + @($(DATA,7,1),HEX)) TO ADDRESS

        * We POKE the last BYTE-PAIR on the line of the HEX file
        * and work our way back to the first BYTE-PAIR on the line.
        DO WHILE COUNT
          STORE STR(ADDRESS+COUNT-1,5)+','+STR(@($(DATA,COUNT*2+8,1),;
            HEX)*16+@($(DATA,COUNT*2+9,1),HEX),3) TO BYTE
          POKE &BYTE
          STORE COUNT -1 TO COUNT
        ENDDO WHILE COUNT

  * GET NEXT LINE OF HEX DATA
  SKIP

ENDDO WHILE $(DATA,2,2) <> '00'
USE
```

End Listing

Simple Calculations With Complex Numbers

by David D. Clark

This article will introduce you to the complex numbers and some of the methods used to perform calculations with them. Despite the title, the ideas involved are quite simple. First, however, a little background on the rational, irrational, and real numbers.

Most people are familiar with the rational numbers, whether they know it or not. Rational numbers are the numbers we use in our everyday lives, like the 25 cents we pay for a newspaper or the 1/2 teaspoon of vanilla extract required for a certain recipe. Rational numbers are those numbers that we can express as the ratio of two integers, say a/b, where the denominator is not equal to zero. The Greek Pythagorean school of mathematicians discovered another fundamental type of number. To their dismay, they found that some quantities cannot be expressed as the ratio of two integers. The first such number they found was $\sqrt{2}$ when they tried to find the number representing the length of the diagonal through a unit square (a square with sides of length one). They showed that no such rational number exists. Soon other examples of these irrational numbers were found. Curiously, π, one of the most intensively studied of nature's fundamental constants, was not shown to be irrational until the eighteenth century. The combined rational and irrational numbers make up the set of real numbers.

David D. Clark, 246 S. Fraser St. #2, State College, PA 16801.

Complex Numbers

The real numbers are themselves a subset of an infinitely larger set of numbers called complex numbers. To understand what complex numbers are, consider the following equation:

$$x^2 + x + 1 = 0 \qquad (1)$$

Equation (1) looks innocuous enough. But, when we apply the quadratic equation in an attempt to solve the equation for the value of x, we obtain:

$$x = 1/2 \pm 1/2\sqrt{3}\sqrt{-1} \qquad (2)$$

What is $\sqrt{-1}$? Good question. Clearly, no real number will work since no real number squared equals -1. Mathematicians have created a special number, usually denoted i or j, to symbolize $\sqrt{-1}$. Thus Equation (2) represents:

"The notation a + bi is a rather unfortunate historical accident. It is better to think of a complex number as an ordered pair."

$$x = 1/2 + 1/2\sqrt{3}\,i \qquad (3)$$
$$x = 1/2 - 1/2\sqrt{3}\,i$$

Such numbers are called complex numbers and consist of a real part (1/2 in Equation (3)) and an imaginary part ($1/2\sqrt{3}\,i$ and $-1/2\sqrt{3}\,i$). They are commonly represented as:

$$x = a + bi \qquad (4)$$

where a and b are real numbers. a is called the real part and bi is called the imaginary part. The real numbers are all of those complex numbers where $b = 0$. Complex numbers where $a = 0$ are called pure imaginaries. Complex

numbers are equal if and only if both the real parts and the imaginary parts are equal. Complex numbers that differ only in the sign of their imaginary parts are called conjugates. An interesting and useful property of conjugates is that, when two conjugates are multiplied, the result is always real.

The notation of Equation (4) is a rather unfortunate historical accident. It tends to confuse people on their first exposure to complex numbers. We can never actually add the real and imaginary parts as the plus sign suggests. It is better to think of a complex number simply as an ordered pair (a, bi). This concept immediately suggests a geometric representation. A plane defined by a real axis and an orthogonal imaginary axis is called an Argand plane after the Swiss-French mathematician Jean Robert Argand (1768 – 1822). The geometric interpretation of complex numbers is also attributed independently to the Norwegian surveyor Caspar Wessel (1745 – 1818). Scholars have suggested that complex numbers be renamed "normal" numbers since, in the geometric representation of such quantities, the imaginary axis is normal to the real axis. Figure 1 (below) shows how the number $3 + 4i$ would be represented on such a plane. Because it is often convenient to think of complex numbers simply as points on a complex plane, we will use the terms *number* and *point* interchangeably in the following discussion.

We can also represent complex numbers with polar coordinates. In polar coordinates, a point is defined in reference to a fixed line and a point on that line, called the origin or pole. In this system, complex numbers are written as:

$$x = r(\cos\theta + i\sin\theta) = r\,\mathrm{Cis}\theta \qquad (5)$$

where r, called the radius or modulus, represents the distance of the point from the origin, and θ, called the amplitude, represents an angle of rotation from the reference line. Figure 2 (below) demonstrates how to graph the number 5 Cis 0.9273. The two forms may be interconverted easily. The polar coordinates for $x = a + bi$ are:

$$r = \sqrt{a^2 + b^2} \qquad (6)$$
$$\theta = \tan^{-1}(a/b)$$

and the rectangular coordinates for $r = \mathrm{Cis}\theta$ are:

$$|a| = r\cos\theta \qquad (7)$$
$$|b| = r\sin\theta$$

Since we calculate only the absolute values of a and b when converting from polar form, we must determine the actual signs of the two quantities from the quadrant in which the point falls. The quadrants are numbered in increasing order starting with quadrant I as the upper righthand part of the graph and proceeding counterclockwise. The table (above) shows how the signs of a and b are related to the quadrant of the plane in which the point appears.

Quadrant	Sign of a	Sign of b
I	+	+
II	−	+
III	−	−
IV	+	−

The signs of the real (a) and imaginary (b) parts of complex numbers as a function of the quadrant in which the point appears.

Table

So, after a little fiddling on a calculator, we see that:

$$3 + 4i = 5\,\mathrm{Cis}\,0.9273 \qquad (8)$$

and that Figures 1 and 2 represent the same number in the two coordinate systems.

Complex Arithmetic

We can do arithmetic with complex numbers just as we do with the more familiar real numbers. The four arithmetic operations of addition, subtraction, multiplication, and division are very simple with complex numbers. As shown below, for multiplication and division we make use of the fact that $i^2 = -1$.

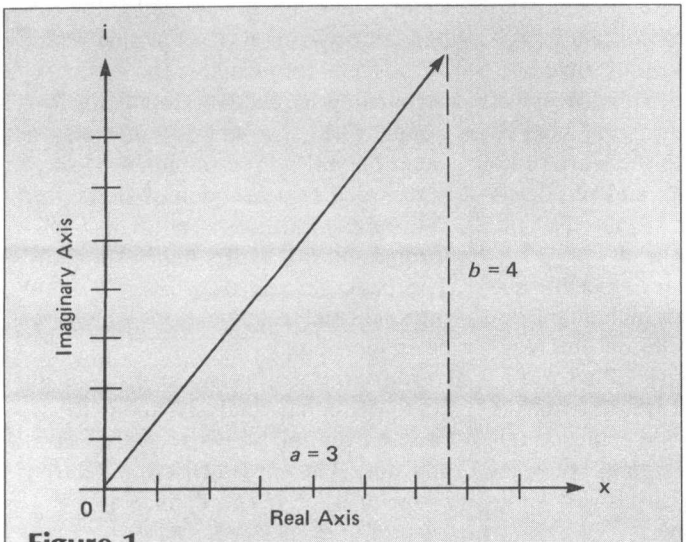


Figure 1.
Geometric representation of the complex number $3 + 4i$ in rectangular coordinates on the Argand plane.

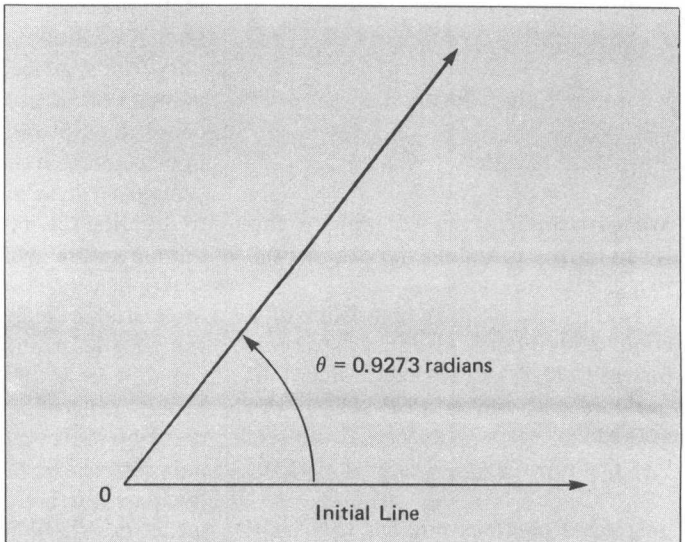


Figure 2.
Geometric representation of the complex number 5 Cisθ graphed in polar coordinates.

Addition

$$(a + bi) + (c + di) = (a + c) + (b + d)i \quad (9)$$

Subtraction

$$(a + bi) - (c + di) = (a - c) + (b - d)i \quad (10)$$

Multiplication

$$(a + bi) \times (c + di) = a(c + di) + b(c + di)i = (ac - bd) + (bc + ad)i \quad (11)$$

Division

$$\frac{a + bi}{c + di} = \frac{(a + bi) \times (c - di)}{(c + di) \times (c - di)} = \frac{ac + bd}{c^2 + d^2} + \left[\frac{bc - ad}{c^2 + d^2}\right] i \quad (12)$$

Complex Functions

The calculation of complex functions is not much more difficult than doing complex arithmetic. One of the simplest complex functions is raising a complex number to an integer power. Polar notation makes the calculation easy:

$$(r\,\mathrm{Cis}\theta)^n = r^n\,\mathrm{Cis}\,n\theta \quad (13)$$

To raise e, the base of the natural logarithms, to a complex power:

$$e^{a + bi} = e^a e^{bi} \quad (14)$$

where, by Euler's formula:

$$e^{bi} = \cos b + i \sin b = \mathrm{Cis}\,b \quad (15)$$

We can confirm the veracity of this formula by expanding the exponential in a Taylor series.

Taking the natural logarithm of a complex number is also fairly straightforward. If θ is restricted such that $-\pi < \theta \leqq \pi$ and $x = r\,\mathrm{Cis}\theta$:

$$\ln(x) = \ln(r) + i\theta \quad (16)$$

To raise a complex number to a complex power, we form an analogy to the operation with real numbers. For real numbers:

$$x^y = e^{y \ln(x)} \quad (17)$$

If x and y are complex:

$$x^y = e^{y \ln(r)}\, e^{iy\theta} \quad (18)$$

Equation (15) suggests a way to calculate the sine and cosine of complex numbers. If x is complex:

$$e^{xi} = \cos x + i \sin x \quad (19)$$

$$e^{-xi} = \cos x - i \sin x \quad (20)$$

Adding Equations (19) and (20) and rearranging, we arrive at:

$$\cos x = \frac{e^{xi} + e^{-xi}}{2} \quad (21)$$

Subtracting Equation (20) from Equation (19) and rearranging yields:

$$\sin x = \frac{e^{xi} - e^{-xi}}{2i} \quad (22)$$

We can now perform a reasonable number of operations with complex numbers. We can build up additional simple functions by proper sequences of the operations described.

Applications

So now that we can do all this wonderful math with complex numbers, what good are they? You might not think so, but complex numbers have a variety of uses. As implied near the beginning of this article, complex numbers have a fundamental importance in the field of mathematics because the roots of polynomial equations often contain imaginary terms. Also, almost all wave phenomena are described with the aid of complex numbers, from signal analysis in electronic circuits to the wave functions used by chemists and physicists to describe the nature of matter at the atomic and molecular level.

Applications familiar to most people interested in electronics include the design and description of filters and reactive and resonant circuits. Phasor diagrams represent the real, resistive parts of the circuit, while the imaginary axis shows the imaginary or reactive part. Such a diagram is very similar to the Argand plane.

When Ohm's law is generalized to AC circuits, it has the conventional form, familiar from DC circuit analysis:

$$I = V/Z$$

where I is the current, V is the voltage, and Z is the impedance. However, the impedance varies depending upon the type of device (resistor, capacitor, or inductor) and the frequency of the signal applied to the circuit:

resistor	$Z_R = R$
capacitor	$Z_C = -i/\omega C$
inductor	$Z_L = i\omega L$

where $\omega = 2\pi f$ and f is the frequency of the applied signal in cycles per second. R is the resistance in ohms, C is the capacitance in farads, and L is the inductance in henrys.

In my own work, I use complex numbers almost daily because of their presence in the Fourier transform. The Fourier transform is most commonly used to convert data in the form of intensity vs. time into frequency vs. time data. I use an instrument called an NMR (Nuclear Magnetic Resonance) spectrometer. This instrument detects the "flipping" of nuclear spin states. This type of experiment originally involved subjecting a sample to a constant radio frequency field and a varying strong magnetic field or a constant magnetic field and varying radio frequency. Varying one of these fields took a lot of time. Modern instruments use the constant magnetic field of a superconducting magnet and a brief, intense pulse of radio energy. Since a square wave, like the applied pulse, can be built up from a series of sine waves of different frequencies, the pulse experiment is equivalent to hitting the sample with all of those frequencies at once. The FID (Free Induction Decay) is recorded as a function of time after the pulse. This data is then transformed into the frequency domain. The peaks in the transformed signal represent the frequencies at which nuclear spins flipped.

Fourier transforms are used in all types of signal analysis applications, such as digital filtering and spectral analysis. The DFT (Discrete Fourier Transform) of $x(n)$ is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) \exp(i2\pi nk/N)$$

$$k = 0,1,\ldots,N-1$$

The inverse DFT is:

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k) \exp(i2\pi nk/N);$$
$$n = 0,1,\ldots,N-1$$

We can also use the Fourier transform to analyze an interesting signal based on a complex function, the frequency-modulated wave related to FM radio transmission:

$$x(k) = \exp[i(ak + b\cos ck)]$$

Analysis of this type of signal reveals a sharp central peak and a number of sidebands whose position and intensities are influenced by the values of the constants *a*, *b*, and *c*.

Complex numbers are also used extensively in the study of electromagnetic phenomena. Calculation of radiated power, radiation pressure, and design of waveguides are good examples of this use.

Computer Implementation

Given all that you can do with complex numbers, the next step is getting your computer to help. It's easy to do in Fortran since COMPLEX is one of the fundamental data types available. In addition, the Fortran library has a large number of functions that operate with COMPLEX variables. However, I have an intense dislike of Fortran, and the Fortran compilers I have available for my microcomputer do not support the COMPLEX data type. Since I *do* like Pascal, I wrote the routines I wanted in that language.

Listing One (page 36) is a UCSD Pascal unit called Complex Arithmetic that implements the operations discussed above. First, a public data type called Complex is declared. Complex numbers will be represented by this data type in programs that make use of the unit. It consists of a record containing two fields. The Re field represents the real coefficient, while the Im field represents the imaginary coefficient. All complex arguments and results are in rectangular coordinates, although some of the calculations use polar coordinates.

This approach has some disadvantages. The primary disadvantage is that you must build up equations from successive procedure and function calls. This is because Pascal does not allow functions to return composite values; it is not possible to write routines that return results of data type Complex. It would also be nice if you could overload the arithmetic operators to use Complex data types. You can do it in Ada, but that's another story.

The four arithmetic operations are straightforward implementations of Equations (9) through (12). The Polar routine, which converts rectangular coordinates to polar coordinates, requires some explanation. Because of the way the p-System handles certain exceptional numerical conditions, a few precautions had to be taken when coding the routine. The first two conditionals are present to prevent underflow when squaring the real and imaginary parts of the argument. Squaring numbers very close to zero can produce a number too close to zero for the computer to represent. For example, squaring 10^{-20} yields 10^{-40}, which is too small for many systems to handle. This is called underflow. When it happens on my machine, a runtime error occurs and the program is aborted. The two conditional statements detect such a situation before the damage is done and prevent an error by setting the coefficient to zero (since Arg is passed by value, the reassignment only has effect while within procedure Polar). Squaring zero does not cause any problems.

The calculation of the amplitude is a little trickier. While the ATan function is very robust (you can't hurt it, no matter what argument you hit it with), division is not quite so resilient. Division can cause problems in a couple of ways. First, division by zero is certain death. Second, dividing a very large number by a very small number can cause overflow. For example, $10^{20}/10^{-20} = 10^{40}$. There is a way to detect such a situation before it happens. Since:

$$\text{abs}(b)/\text{abs}(a) = e^{(\ln(\text{abs}(b)) - \ln(\text{abs}(a)))}$$

we can check the relative magnitudes of *a* and *b* before doing the actual division. If an infinite quantity (to the computer) is about to be calculated, signs are checked and an appropriate value is assigned directly. (The ATan function approaches $\pi/2$ as its argument increases without bound in the positive direction. It tends to $-\pi/2$ as the argument decreases toward negative infinity.) All this checking makes Polar an expensive procedure in terms of execution time. If you don't need all of these safeguards, you can reduce the procedure to the two statements:

```
Modulus :=
    Sqrt(Sqr(Arg.Re) + Sqr
      (Arg.Im));

Amplitude :=
    ATan(Arg.Im/Arg.Re);
```

(A note to p-System users: Some early versions of the interpreter have Ln functions that do not always return a result of the correct sign.)

The CToPower, CExp, and CLn procedures are almost straight transliterations of Equations (13) through (16). The procedures CToC, CSin, and CCos are examples of how equations involving complex numbers can be built up one operation at a time. Other functions, such as taking complex powers of real numbers, can be built up by calling these procedures with appropriate arguments. For example, to raise 5.3 to the $7.0 + 3.0i$ power, use CToC with Arg1 $= 5.3 + 0.0i$ and Arg2 $= 7.0 + 3.0i$.

Listing Two (page 42) is a typical application for complex numbers, a test program for FFT (Fast Fourier Transform) functions. The program generates a function with an analytically known DFT, transforms it, compares the calculated transform with the analytic transform, then reverses the process. The calculated inverse transform is then compared with the original function. The function generated is:

$$x(n) = Q^n \quad n = 0,1,\ldots,N-1$$

and its DFT is:

$$X(k) = (1 - Q^N)/(1 - QW^k)$$
$$k = 0,1,\ldots,N-1$$

where $W = \exp(-i2\pi/N)$ and Q is the complex constant $0.9 + 0.3i$. In the program, $N = 64$ so the transform is performed on 64 complex points.

The procedure that performs the actual calculation of the FFT is "included" from the file CURFFT.TEXT,

shown in Listing Three (page 46). That way you can test several different FFT routines using the same main program. Just create your new Fourea procedure and stuff it in a file called CURFFT.TEXT. As stated in the listing, this is not a particularly efficient FFT procedure, but it has the virtue of simplicity (for an FFT calculation anyway) and illustrates a "real life" application of complex numbers.

The output at the end of Listing Three shows the results of running the FFT testing program. A large difference between the calculated transforms and the analytically known values for the transforms indicates that there is probably an error in the Fourea procedure being tested.

Summary

Complex numbers have been introduced in relation to the more familiar rational and real number systems. Complex numbers consist of a real part and an imaginary part. The imaginary part is so named because one of its factors is $\sqrt{-1}$, more commonly denoted simply as *i*. A geometrical interpretation of such numbers has been described in which complex numbers represent points on a complex plane, called an Argand plane. The points can be plotted in rectangular or polar coordinates, and the two coordinate systems can be easily interconverted.

Complex numbers have an associated algebra, just like the rational and real numbers. The operations of addition, subtraction, multiplication, and division, as well as the calculation of some common functions of complex numbers, have been explained. A UCSD Pascal unit called Complex Arithmetic, which implements these calculations in a specific programming language, has been described, as well as an example program to calculate fast Fourier transforms using complex arithmetic.

In addition to signal analysis and Fourier analysis, complex numbers are used in the mathematics of electronics (e.g., filters and reactive and resonant circuits). Phasor diagrams consist of a real part, the resistive part, and an imaginary part, the reactive part. The wave functions used by chemists and physicists in the study of quantum mechanics also make use of the properties of complex numbers. And now, so can you.

References

1. Crandall, R. E., *Pascal Applications for the Sciences* (John Wiley & Sons, Inc., 1984). A very good "how to" book on the use of computers in the sciences. It contains a large number of small Pascal programs for solving problems in several areas of the physical and biological sciences.
2. Dence, J. B., *Mathematical Techniques in Chemistry* (John Wiley & Sons, Inc., 1975). This book is invaluable to chemists. It describes a wide range of mathematical techniques as applied to chemical problems.
3. Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, *Programs for Digital Signal Processing* (IEEE Press, 1979). This book contains lots of very fancy programs for use in various areas of signal processing. The discussion of the theory of such calculations is minimal. All the programs are in Fortran. The program in Listings Two and Three was derived from a simple demonstration program in this book.
4. Horowitz, P., and W. Hill, *The Art of Electronics* (Cambridge University Press, 1980). This book contains a good introduction to the use of complex quantities in the design and description of electronic signal processing circuitry such as filters and resonant and power circuits. It is also an excellent introduction to electronics in general.
5. Kousourou, G., S. N. Sonsky, and F. Scalzo, *An Introduction to Technical Mathematics with Computing* (Petrocelli Books, Inc., 1979). A textbook of elementary mathematics required in most technical work. It has a good introduction to complex numbers and their manipulation. Lots of problems and worked examples.
6. Lorrain, P., and D. R. Corson, *Electromagnetic Fields and Waves* (W. H. Freeman and Company, San Francisco, 1970). Some very heavy reading on the theoretical aspects of electromagnetic phenomena. Complex quantities permeate the mathematics of this book.

DDJ

(Listings begin on page 36)

Calculations (Text begins on page 30)

Listing One

```
{xL PRINTER:}
UNIT ComplexArithmetic;
{ data type and procedures for performing complex arithmetic }

  INTERFACE

    TYPE  Complex = RECORD
                      Re : Real;
                      Im : Real
                    END { of RECORD };

    PROCEDURE CAdd(VAR Result: Complex; Arg1, Arg2: Complex);
    { adds "Arg1" and "Arg2" and returns the sum in "Result" }

    PROCEDURE CSub(VAR Result: Complex; Arg1, Arg2: Complex);
    { subtracts "Arg2" from "Arg1" and returns the difference in "Result" }

    PROCEDURE CMult(VAR Result: Complex; Arg1, Arg2: Complex);
    { multiplies "Arg1" and "Arg2" and returns the product in "Result" }

    PROCEDURE CDiv(VAR Result: Complex; Arg1, Arg2: Complex);
    { divide "Arg1" by "Arg2" and returns the quotient in "Result" }

    PROCEDURE Polar(Arg: Complex; VAR Modulus, Amplitude: Real);
    { converts a Complex number "Arg" in rectangular form to Polar form }

    PROCEDURE CToPower(VAR Result: Complex; Arg: Complex; Power: Integer);
    { raises "Arg" to the positive integral "Power" and returns the
      answer in "Result" }

    PROCEDURE CExp(VAR Result: Complex; Arg: Complex);
    { raises e to the "Arg" and returns the answer in "Result" }

    PROCEDURE CLn(VAR Result: Complex; Arg: Complex);
    { takes the natural logarithm of "Arg" and returns the answer in
      "Result" }

    PROCEDURE CToC(VAR Result: Complex; Arg1, Arg2: Complex);
    { raise Complex number "Arg1" to Complex Power "Arg2" }

    PROCEDURE CSin(VAR Result: Complex; Arg: Complex);
    { takes the sine of "Arg" and returns it in "Result }

    PROCEDURE CCos(VAR Result: Complex; Arg: Complex);
    { takes the cosine of "Arg" and returns it in "Result }


  IMPLEMENTATION

    CONST LN_MAX_REAL = 87.49823353; { ln(1.0e38) }
          PI_OVER_2   = 1.570796327; { pi/2.0 }
          CLOSEST     = 1E-19;       { sqrt(1.0e-38 }
PROCEDURE CAdd{VAR Result: Complex; Arg1, Arg2: Complex};

BEGIN { CAdd }
  Result.Re := Arg1.Re + Arg2.Re;
```

(Continued on next page)

Listing One

```
  Result.Im := Arg1.Im + Arg2.Im
END { of CAdd };


PROCEDURE CSub{VAR Result: Complex; Arg1, Arg2: Complex};

BEGIN { CSub }
  Result.Re := Arg1.Re - Arg2.Re;
  Result.Im := Arg1.Im - Arg2.Im
END { of CSub };


PROCEDURE CMult{VAR Result: Complex; Arg1, Arg2: Complex};

BEGIN { CMult }
  Result.Re := Arg1.Re*Arg2.Re - Arg1.Im*Arg2.Im;
  Result.Im := Arg1.Im*Arg2.Re + Arg1.Re*Arg2.Im
END { of CMult };


PROCEDURE CDiv{VAR Result: Complex; Arg1, Arg2: Complex};

VAR   Denom: Real;

BEGIN { CDiv }
  Denom := Sqr(Arg2.Re) + Sqr(Arg2.Im);
  Result.Re := (Arg1.Re*Arg2.Re + Arg1.Im*Arg2.Im)/Denom;
  Result.Im := (Arg1.Im*Arg2.Re - Arg1.Re*Arg2.Im)/Denom
END { of CDiv };


PROCEDURE Polar{Arg: Complex; VAR Modulus, Amplitude: Real};

BEGIN { Polar }
  WITH Arg DO BEGIN
       IF Abs(Re) < CLOSEST THEN
          Re := 0.0;
       IF Abs(Im) < CLOSEST THEN
          Im := 0.0;
       Modulus := Sqrt(Sqr(Re) + Sqr(Im));
       IF Im = 0.0 THEN
          Amplitude := 0.0
       ELSE IF Re = 0.0 THEN
          IF Im > 0.0 THEN
             Amplitude := PI_OVER_2
          ELSE
             Amplitude := -PI_OVER_2
       ELSE IF (Ln(Abs(Im)) - Ln(Abs(Re)) > LN_MAX_REAL) THEN
          IF Re > 0.0 THEN
          IF Im > 0.0 THEN
             Amplitude := PI_OVER_2
          ELSE
             Amplitude := -PI_OVER_2
       ELSE
          IF Im > 0.0 THEN
             Amplitude := -PI_OVER_2
          ELSE
             Amplitude := PI_OVER_2
    ELSE
```

```
        Amplitude := ATan(Im/Re)
  END { of WITH Arg }
END { of Polar };


PROCEDURE CToPower{VAR Result: Complex; Arg: Complex; Power: Integer}

VAR   I: Integer;
      Modulus, Amplitude, NewMod, PowAmp: Real;

BEGIN { CToPower }
  IF Power = 0 THEN BEGIN
     Result.Re := 1.0;
     Result.Im := 0.0
  END
  ELSE BEGIN
       Polar(Arg, Modulus, Amplitude);
       NewMod := 1;
       IF Power > 0 THEN
          FOR I := 1 TO Power DO NewMod := NewMod*Modulus
       ELSE
          FOR I := 1 TO Abs(Power) DO NewMod := NewMod/Modulus;
       PowAmp := Power*Amplitude;
       Result.Re := NewMod*Cos(PowAmp);
       Result.Im := NewMod*Sin(PowAmp)
  END
END { of CToPower };


PROCEDURE CExp{VAR Result: Complex; Arg: Complex};

VAR   Expo: Real;

BEGIN { CExp }
  Expo := Exp(Arg.Re);
  Result.Re := Expo*Cos(Arg.Im);
  Result.Im := Expo*Sin(Arg.Im)
END { of CExp };


PROCEDURE CLn{VAR Result: Complex; Arg: Complex};

VAR   Modulus, Amplitude: Real;

BEGIN { CLn }
  Polar(Arg, Modulus, Amplitude);
  Result.Re := Ln(Modulus);
  Result.Im := Amplitude
END { of CLn };


PROCEDURE CToC{VAR Result: Complex; Arg1, Arg2: Complex};

VAR   LogPart, Expo: Complex;

BEGIN { CToC }
  CLn(LogPart, Arg1);
  CMult(Expo, Arg2, LogPart);
  CExp(Result, Expo)
END { of CToC };
```

(Continued on next page)

```
    PROCEDURE CSin{VAR Result: Complex; Arg: Complex};

    VAR Expl, Exp2, Partl, Part2, Sum, Divisor: Complex;

    BEGIN { CSin }
      Expl.Re := -Arg.Im;          { z*i }
      Expl.Im := Arg.Re;
      CExp(Partl, Expl);           { exp(zi) }
      Exp2.Re := Arg.Im;           { -z*i }
      Exp2.Im := -Arg.Re;
      CExp(Part2, Exp2);           { exp(-zi) }
      CSub(Sum, Partl, Part2);     { exp(zi) - exp(-zi) }
      Divisor.Re := 0.0;
      Divisor.Im := 2.0;
      CDiv(Result, Sum, Divisor) { (exp(zi) - exp(-zi))/(2i) }
    END { of CSin };


    PROCEDURE CCos{VAR Result: Complex; Arg: Complex};

    VAR Expl, Exp2, Partl, Part2, Sum, Divisor: Complex;

    BEGIN { CCos }
      Expl.Re := -Arg.Im;          { z*i }
      Expl.Im := Arg.Re;
      CExp(Partl, Expl);           { exp(zi) }
      Exp2.Re := Arg.Im;           { -z*i }
      Exp2.Im := -Arg.Re;
      CExp(Part2, Exp2);           { exp(-zi) }
      CAdd(Sum, Partl, Part2);     { exp(zi) + exp(-zi) }
      Divisor.Re := 2.0;
      Divisor.Im := 0.0;
      CDiv(Result, Sum, Divisor) { (exp(zi) + exp(-zi))/(2) }
    END { of CCos };


BEGIN { ComplexArithmetic }
END { of ComplexArithmetic }.
```

End Listing One

(Listing two begins on next page)

Listing Two

```
Program FFTTest(Input, Output, Prn);

{*
**      This program generates a function to test various fast Fourier
**  transform programs.  As currently implemented, the program compiles the
**  routine to be tested in-line with the test program.  This is
**  accomplished through the use of the "include" file mechanism of the
**  compiler.  The routine to be tested should be in a file called
**  "CURFFT.TEXT".  The routine's heading should be as follows:
**
**      Procedure Fourea(    numPoints : Integer;
**                           whichWay  : Direction;
**                       Var f         : CmplxArray);
**
**  where the types Direction and CmplxArray are as declared below.
**
**      The program computes the function A^I, I = 0, 1, ... MAXPOINTS - 1.
**  The discrete Fourier transform of this function is also computed.  The
**  procedure Fourea is called to compute the fast Fourier transform in the
**  forward direction.  This call is then repeated, but for the reverse
**  direction.  The theoretical transform is then compared to the computed
**  transform and the calculated reverse transform is compared to the
**  original function.  The maximum difference of each comparison is then
**  printed.  A large maximum difference probably indicates a program error.
**
**      The Unit ComplexArithmetic is used to perform the arithmetic on
**  complex numbers required by the algorithm.  The value of MAXPOINTS is
**  set at 64 because it is also a power of 8.  This is important because
**  the program is also intended to test a version of Fourea optimized for
**  data arrays of 512 complex points.
**
**  Based on a Fortran program by C. M. Rader
**  MIT Lincoln Laboratory, Lexington MA   02173
**
**  Written by David D. Clark
**  26-Mar-83
*}

Uses {$U MATH.LIBRARY} ComplexArithmetic;

Const MAXPOINTS    = 64;                  { size of the transformed array }
      TWOPI        = 6.2831853;           { 2 * pi }
      PRINTFILE    = 'PRINTER:';          { output device file name for results

Type  Direction    = (Forwurd, Reverse); { note the spelling }
      CharFile     = File of Char;
      CmplxArray   = Array [1..MAXPOINTS] of Complex;

Var   I            : Integer;             { general purpose index variable }
      DD,
      D1,
      D2,
      GG,
      G1,
      G2,

TwoPiDivMax : Real;                       { will be TWOPI/MAXPOINTS }
OneZero,                                  { 1.0 + 0.0*i (complex number 1) }
A,
D,
```

(Continued on next page)

Listing Two

```
      E,
      W,
      Tem,
      TemTem      : Complex;
      B,                                  { complex data arrays }
      C,
      Qb          : CmplxArray;
      PrnFile     : CharFile;             { print out file }


   Procedure PrintOut(   numPoints : Integer;
                      Var f        : CmplxArray;
                      Var dev      : CharFile   );
   {*
   **  Print out the numPoint members of complex array f on dev
   *}

   Var   i,
         j : Integer;

   Begin { PrintOut }
      WriteLn(dev);
      For i := 1 to numPoints Div 2 Do Begin
          j := 2*i - 1;
          WriteLn(dev, '(', j      : 2, ')', f[j    ].re: 14, f[j    ].im: 14,
                       '    (', (j + 1): 2, ')', f[j + 1].re: 14, f[j + 1].im: 14)
      End;
      WriteLn(dev);
      WriteLn(dev)
   End { of PrintOut };


{ now include the FFT procedure }
{$I CURFFT.TEXT}


Begin { FFTTest }

   { some initialization }
   OneZero.re := 1.0;  OneZero.im := 0.0;
   TwoPiDivMax := TWOPI/MAXPOINTS;
   W.re := Cos(TwoPiDivMax);  W.im := -Sin(TwoPiDivMax);
   A.re := 0.9;  A.im := 0.3;
   ReWrite(PrnFile, PRINTFILE);

   { Calculate and print function A^I, I := 0, 1, ... (MAXPOINTS - 1) }
   B[1] := OneZero;
   Qb[1] := OneZero;
   For I := 2 to MAXPOINTS Do Begin
       CToPower(B[I], A, I - 1);
       Qb[I] := B[I]

End;
WriteLn(PrnFile, 'Complex input sequence:');
PrintOut(MAXPOINTS, Qb, PrnFile);

{ Calculate and print the theorectical discrete Fourier transform }
CToPower(Tem, A, MAXPOINTS);
CSub(D, OneZero, Tem);
For I := 1 to MAXPOINTS Do Begin
```

```
    CToPower(TemTem, W, I - 1);
    CMult(Tem, A, TemTem);
    CSub(E, OneZero, Tem);
    CDiv(C[I], D, E)
End;
WriteLn(PrnFile, 'Theoretical discrete Fourier transform:');
PrintOut(MAXPOINTS, C, PrnFile);

{ Calculate and print the fast Fourier transform of the function }
Fourea(MAXPOINTS, Forwurd, B);
WriteLn(PrnFile, '"Fourea" generated discrete Fourier transform:');
PrintOut(MAXPOINTS, B, PrnFile);

{ Find the maximum difference }
DD := 0.0;
For I := 1 to MAXPOINTS Do Begin
    D1 := Abs(C[I].re - B[i].re);
    D2 := Abs(C[I].im - B[I].im);

    If D1 > DD Then
       If D2 > D1 Then
          DD := D2
       Else
          DD := D1
    Else If D2 > DD then
       DD := D2

End;

{ Calculate and print the inverse transform }
Fourea(MAXPOINTS, Reverse, C);
WriteLn(PrnFile, '"Fourea" generated inverse descrete Fourier transform:');
PrintOut(MAXPOINTS, C, PrnFile);

{ Find the maximum difference }
GG := 0.0;
For I := 1 to MAXPOINTS Do Begin
    G1 := Abs(Qb[I].re - C[i].re);
    G2 := Abs(Qb[I].im - C[I].im);

    If G1 > GG Then
       If G2 > G1 Then
          GG := G2
       Else
          GG := G1
    Else If G2 > GG then
       GG := G2

   End;

   { Print out the maximum differences }
   WriteLn(PrnFile, 'Max diff between theor. and Fourea DFT is ', DD);
   WriteLn(PrnFile, 'Max diff between orig. and inverse is ', GG)

End { of FFTTest }.
```

End Listing Two

(Continued on next page)

Listing Three

```
Procedure Fourea(     numPoints : Integer;
                      whichWay  : Direction;
                  Var f         : CmplxArray);

{*
**       Performs a Cooley-Tukey type fast Fourier transform.
**
**       f is a one dimensional complex array whose length, numPoints, is
**  a power of two. whichWay determines in which direction the transform
**  will be performed. If it is Forwurd, then isInverse is set to -1 and
**  a forward transform is carried out. If whichWay has a value of Reverse,
**  then isInverse is set to 1 and a reverse transform is calculated.
**
**       transform[j] := sum(f[i]*w^((i-1)*(j-1))), where i and j run from
**  1 to numPoints and w := exp(isInverse*2*pi*sqrt(-1)/numPoints). The
**  program also computes the inverse transform, for which the defining
**  expression is: inverse DFT := (1/numPoints)*sum(f[i]*w^((i-1)*(j-1))).
**
**       Run time is proportional to numPoints*Log2(numPoints) rather than
**  to numPoints^2 for the classical discrete Fourier transform.
**
**       This is a very short version of the FFT and is only intended for
**  demonstration purposes. Programs are available which run faster and
**  are not restricted to numbers of points that are powers of two or to one
**  dimensional arrays.
*}

Var   isInverse : Integer;


   Function PowerOfTwo(    numPoints : Integer) : Boolean;
   {*
   **  Determine if numPoints is an integer power of 2.
   *}

   Var   modulo : Integer;

   Begin { PowerOfTwo }
      PowerOfTwo := True;
      modulo := 0;
      While (modulo = 0) and (numPoints >= 2) Do Begin
            modulo := numPoints Mod 2;
            If modulo = 0 Then
               numPoints := numPoints Div 2
            Else
               PowerOfTwo := False
      End
   End { of PowerOfTwo };


   Procedure Scramble(     numPoints : Integer;
                       Var f         : CmplxArray);
   {*
   **  Put the data in bit reversed order.
   *}
 Var   i,
       j,
       m    : Integer;
       temp : Complex;
```

```
Begin { Scramble }
   j := 1;
   For i := 1 to numPoints Do Begin
       If i < j Then Begin
          temp := f[j];
          f[j] := f[i];
          f[i] := temp
       End;
       m := numPoints Div 2;
       If m < j Then
          While m < j Do Begin
                j := j - m;
                m := (m + 1) Div 2
          End;
       j := j + m
   End
End { of Scramble };


Procedure Butterflies(     numPoints,
                           isInverse : Integer;
                       Var f         : CmplxArray);
{*
**  Calculate the butterflies for the bit reversed data of an FFT.
**  Normalize if a reverse FFT is performed.
*}

Const pi    = 3.1415927;

Var   mMax,
      step,
      index,
      m,
      i,
      j     : Integer;
      theta : Real;
      w,
      cn,
      temp  : Complex;

Begin { Butterflies }
   mMax := 1;
   While numPoints > mMax Do Begin
         step := 2*mMax;
         For m := 1 to mMax Do Begin
             theta := pi*(isInverse*(m - 1))/mMax;
             w.re := Cos(theta);  w.im := Sin(theta);
             For i := 1 to ((numPoints - m) Div step) + 1 Do Begin
                 index := m + ((i - 1)*step);
                 j := index + mMax;
                 CMult(temp, w, f[j]);
                 CSub(f[j], f[index], temp);
                 CAdd(f[index], f[index], temp)
             End
         End;
         mMax := step
   End;
   If isInverse = 1 Then Begin
      cn.re := numPoints;  cn.im := 0.0;
      For i := 1 to numPoints Do
```

(Continued on next page)

Listing Three

```
                CDiv(f[i], f[i], cn)
         End
      End { of Butterflies };
Begin { Fourea }
   WriteLn('Fourea');
   If PowerOfTwo(numPoints) Then Begin
      If whichWay = Forwurd Then
         isInverse := -1
      Else
         isInverse := 1;
      Scramble(numPoints, f);
      Butterflies(numPoints, isInverse, f)
   End
   Else Begin
      WriteLn('The number of points in the array was not a power of two.');
      WriteLn('No transformation performed.')
   End
End { of Fourea };
```

"Truncated Output from the Program"

```
Complex input sequence:

( 1)       1.00000        0.00000     ( 2)    9.00000E-1    3.00000E-1
( 3)    7.20000E-1     5.40000E-1     ( 4)    4.86000E-1    7.02000E-1
           .              .                      .             .
           .              .                      .             .
           .              .                      .             .
(61)    3.80692E-2     1.86474E-2     (62)    2.86681E-2    2.82034E-2
(63)    1.73402E-2     3.39835E-2     (64)    5.41117E-3    3.57872E-2


Theoretical discrete Fourier transform:

( 1)       1.10736        2.98377     ( 2)       1.65441       4.19275
( 3)       3.60049        6.69406     ( 4)     1.58381E1       7.26182
           .              .                      .             .
           .              .                      .             .
           .              .                      .             .
(61)    6.55765E-1        1.31934     (62)    6.98674E-1       1.54663
(63)    7.65931E-1        1.85433     (64)    8.81321E-1       2.29593


"Fourea" generated discrete Fourier transform:

( 1)       1.10736        2.98377     ( 2)       1.65441       4.19275
( 3)       3.60051        6.69405     ( 4)     1.58380E1       7.26169
           .              .                      .             .
           .              .                      .             .
           .              .                      .             .
(61)    6.55781E-1        1.31934     (62)    6.98693E-1       1.54663
(63)    7.65960E-1        1.85433     (64)    8.81363E-1       2.29592


"Fourea" generated inverse descrete Fourier transform:

( 1)    9.99996E-1     1.02073E-6     ( 2)    8.99998E-1    3.00002E-1
( 3)    7.19999E-1     5.40002E-1     ( 4)    4.85999E-1    7.02002E-1
           .              .                      .             .
           .              .                      .             .
           .              .                      .             .
(61)    3.80697E-2     1.86473E-2     (62)    2.86679E-2    2.82035E-2
(63)    1.73393E-2     3.39839E-2     (64)    5.40911E-3    3.57877E-2


Max diff between theor. and Fourea DFT is  1.24931E-4
Max diff between orig. and inverse is  3.81470E-6
```

End Listings

GREP.C

A Unix-Like, Generalized, Regular Expression Parser in C

by Allen Holub

"The power of grep lies in its use of regular expressions as pattern templates rather than explicit strings."

Grep is the Unix pattern finder: it goes into a file or group of files and finds text patterns matching a symbolic regular expression. Grep is surprisingly useful. With it you can find a subroutine lost in one of the 50 modules making up the giant program that you're working on. You can find the misspelled name that your linker says is an undeclared function. Grep can number all the lines in a file or list all procedure declarations in a C program, as well as perform any of innumerable other things.

A good example of grep's utility is the history of this version, grep.c. I started out wanting to expand the pattern-searching capabilities of Ed Reams' editor, RED (*DDJ* No. 81). I wanted to add a pattern-searching capability similar to that of the Berkeley Unix editor, vi. So I set about converting into C the pattern-matching algorithms in *Software Tools in Pascal* by Kernighan and Plauger (Addison-Wesley, 1981, Ch. 5).

Along the way I ran into difficulties. My version of RED was written in BDS C, which has a nonstandard I/O library. I wanted to translate the editor over to standard C so I could port it between different compilers and different machines. To do the translation, I needed to find all calls to the nonstandard library routines. So I turned my pattern matcher into a real program, linked the BDS version of RED without linking the library modules (to get a list of the library routines that RED used), then used grep to search all the modules of RED for procedure calls to the library functions.

The program presented in this article is most of the Unix grep. The only omissions are those command line switches that are Unix dependent, the −x switch (which performs an exact line match), and the +, ?, and ( ) regular expression operators (which are not essential).

The power of grep lies in its use of regular expressions as pattern templates rather than explicit strings. For example, the grep command line:

```
grep ^[a - z][a - z]*[\s\t]*.*([^;]*)[^;]*$
     mod1.c mod2.c mod3.c
```

creates a cross reference of a large C program. The three files mod1.c, mod2.c, and mod3.c are searched. Grep's output will show all subroutine declarations along with the name of the file in which the subroutine is declared. The regular expression is interpreted as follows: beginning of line (^), followed by one or more occurrences of any character in the range a to z ([a −⇅ z][a − z]*), followed by either a space or a tab repeated zero or more times ([\s\t]*), followed by any character repeated zero or more times (.*), followed by an open parenthesis (( ), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by a close parenthesis ( )), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by end of line ($).

Allen Holub, Software Engineering Consultants, P.O. Box 5679, Berkeley, CA 94705.

Copyright © 1983, 1984 by Allen Holub. All rights reserved. Permission is granted for personal, non-commercial use only. Any use for profit or other commercial gain without written permission of the author is prohibited.

You could also use grep to get a count of the number of procedures found and either have the matching lines printed out with line numbers or have all the lines not matching the pattern printed by including various command line switches in the program invocation.

Regular Expressions

Regular expressions are a way of representing text patterns in a symbolic shorthand. The * and ? that CP/M uses are examples of a crude regular expression syntax. The symbols grep uses to define regular expressions fall into five categories:

- Symbols that match a specific character
- Symbols that match any character
- Symbols that match a character's position on the line
- Symbols (called "character classes") that match any of a set of characters or anything except a set of characters
- Symbols that let you match the previous symbol any number of times (called "closure")

The rules for constructing regular expressions are given on the excerpted manual page (Figure 1, page 53). Some examples follow.

a.d
matches any word containing an "a," followed by any character, followed by a "d." This expression will match the substrings "and" in "*and*" and "ard" in "a*ard*vark"; this means that grep will print any *line* containing either word, along with any other match.

^a.d
will match the same strings but only if they occur at the beginning of the line. No characters, including spaces and tabs, are allowed in front of the "a."

a.d$
will match the same strings if they occur at the end of the line. No character, including spaces and tabs, can follow the "d."

^$
will match a beginning of line, followed by an end of line; this means that it will match all lines containing nothing but a newline character.

an*d
will match any word containing an "a," followed by an "n" repeated zero or more times. This expression will match "add" as well as "and."

.*
will match any character repeated any number of times. This expression will always succeed. For example, an invocation of grep with the line

grep −n .* (filename)
will output every line in the file preceded by its line number.

aa*
will match one (rather than zero) or more occurrences of the letter "a." For example,

[abc]
defines a "character class." A character class matches any one of the characters surrounded by the square brackets. This particular character class will match an "a," "b," or "c" in the corresponding position on the line.

who[ms]e
matches "who," followed immediately by either an "m" or an "s," followed by an "e." That is, the words "whomever" and "whose" will be matched, but the word "whole" will not. Character class definitions may be abbreviated by using a dash. For example,

[a – k]
will be treated as if you had said [abcdefghijk]. Similarly,

[0 – 9A – Fa – f]
will be expanded to [0123456789ABCDEFabcdef]. This last character class will match any single hexadecimal digit. That is, any digit or any letter between "a" and "f" will be matched. Note that you have to say "A – Fa – f" to match both upper and lower case.

0x[0 – 9a – fA – F][0 – 9a – fA – f]*
will match all lines in a C program that contain hexadecimal numbers. This regular expression matches the characters 0x, followed by any of the characters 0123456789abc defABCDEF repeated one or more times. A "negative character class" (one that matches any character except those listed) may be defined by using ^ as the first character following the [. For example,

f[^o]
will find all occurrences of an "f" not followed by an "o." Be careful here with patterns at the end of line. Although [^o] matches any character that is not an "o," a lone "f" at the end of line will not match the pattern because the end of line is not a character – it is a position. f[^o]*$ or f$|f[^o] will find an "f" at the end of line.

[a – zA – Z]f\s|[A – Z][A – Z]*\s
will match all lines containing words ending in "f" or all lines containing words composed only of upper-case characters. That is, if either of the regular expressions separated by the | are satisfied, a match is returned.

If you need to match a character that is used as a symbol in the regular expression, precede it with a backslash (\). For example, * will match an asterisk, and \\ will match the backslash itself. Certain escape sequences (as these backslash sequences are called) are predefined; in particular, \s matches a space and \t matches a tab (control-I).

These are needed because of the irregularities of certain compilers and operating systems. A space in the command line will make many command-line interpreters break up the expression into two arguments. A tab in the command line will confuse CP/M utterly: it won't execute your program at all. Other escape sequences are defined on the manual page (Figure 1 below).

Technical Description

The routines in tools.c differ from those in *Software Tools* in three important ways. First, the routines were translated into C. Second, all references to array indexes were replaced with pointers in the interest of increased execution speed. Finally, the data structure used for the pattern template was changed significantly.

The reasons for this last change are somewhat complex. Grep breaks up the input expression into a pattern template, where each element of the template represents a single logical portion of the expression. For example, the expression

[a – z]x.*

requires four elements in the pattern template. One element is required for the character class ("[a – z]"), one element for the literal character match ("x"), one element to match any character (the ".") and one element for the closure (the "*"). Processing the expression is much easier once it has been functionally divided in this way.

Kernighan and Plauger use a single ASCII string as their pattern template, and this data structure causes several problems. Varying numbers of characters are required to represent different types of elements in the template. To advance through the template, you need a subroutine that analyzes the current element and then advances the appropriate number of characters; this subroutine adds unnecessary overhead to the pattern-recognizing parts of grep. By replacing the ASCII string with a linked list of structures (which is how the templates are represented in my version of grep), you can advance to the next pattern with a single assignment operation.

Grep can be broken up into three distinct parts:

(1) Get the regular expression(s), the file list, and any switches from the command line.
(2) Translate the expression(s) into a pattern template.
(3) Go through the input files one line at a time, calling the routine matchs( ), and produce the appropriate output on finding a match.

Getting the Expressions

Grep is divided into two main modules: grep.c and tools.c. Grep.c does all of the I/O and tools.c contains the pattern-matching routines. Grep translates the pattern strings into a special template representing the pattern (more about this later), while the routine matchs( ) does the actual pattern matching; it processes all symbols except the OR (|) operator, which separates the regular expressions.

Grep creates a template for every regular expression input and organizes these templates using an array of pointers to templates (similar to argv) called exprv[ ]; a count of the number of separate expressions in the array (exprc) is also available. Grep then calls matchs( ), once for each template in exprcv[ ], before getting the next input line.

The Pattern Template

The templates are a linked list of structures called TOKENs

NAME

grep—search a file for a pattern

SYNOPSIS

grep [–options] . . . [expression] [filelist] . . .

DESCRIPTION

This program will find a string specified by a regular expression in a file or group of files. The following options are recognized:

–v All lines but those matching are printed.
–c Only a count of the matching lines is printed.
–l The names of the files with matching lines are listed (once) separated by newlines.
–n Each line is preceded by its line number in the file.
–h Do not print filename headers with output lines.
–y All characters in the file are mapped to upper case before matching. This is the default if the regular expression is given on the command line (because CP/M maps everything on the command line to upper case). Use the –f option if you need both lower and upper case.
–e <expression> Same as a simple expression argument, but useful when the expression begins with a "–."
–f <file> The regular expression is taken from the file. If several regular expressions are listed (separated by newlines or | s) then a match will be flagged if any of the regular expressions are satisfied. –e and –f are mutually exclusive. If –f is given, any regular expression on the command line is taken to be a filename.

Regular expressions are composed of the following:

A ^ matches the beginning of a line.

A $ matches the end of a line.

A \ followed by a single character matches that character. In this way a "*" will match an asterisk, a "\." matches a period, etc. The following sequences are special:

Figure 1

\b backspace (^H)
\n linefeed (^J this is not the same as $)
\r carriage return (^M)
\s space
\t tab (^I)
\\ backslash

A . matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [ ] specifies a "character class." Any single character in the string will be matched. For example "[abc]" will match an a, b, or c. Ranges of ASCII character codes may be abbreviated as in "[a–zO–9]." If the first symbol following the [is a ^ then a "negative character class" is specified. In this case, the string matches all characters except those enclosed in the brackets (i.e., [^a–z] matches everything except lower case letters). Note that a negative character class must match something, even though that something cannot be any of the characters listed. For example: "^$" is not the same as "^[^z]$." The first example will match an empty line (beginning of line followed by end of line); the second example matches a beginning of line followed by any character except a z followed by end of line. In the second example a character must be present on the line, but that character can't be a z. Note that *, ., ^, and $ are not special characters when inside a character class.

A regular expression followed by a * matches zero or more matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by a | or a newline match either a match for the first or a match for the second.

The order of precedence is [ ] then * concatenation then | then newline.

EXAMPLE:

The command line:

grep –n ^[a–z] [a–z] * [\s\t] *.* ([^;]*) [^;]*$ <file list>

creates a cross reference of a large C program. "<file list>" should be replaced with a list of the modules to be searched. Grep's output will show all subroutine declarations in all the listed files. In addition, every output line will be preceded by both the name of the file in which the line was found (this is automatic if more than one file is searched) and by the appropriate line number (the –n causes line numbers to be shown).

The regular expression is interpreted as follows: beginning of line (^) followed by one or more occurrences of any character in the range a to z ([a–z] [a–z]*), followed by either a space or a tab repeated zero or more times ([\s\t]*), followed by any character repeated zero or more times (.*), followed by a open parenthesis (( ), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by a close parenthesis ( )), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by end of line ($).

BUGS

All features of the unix version of grep are supported except that the –s, –x, and –b options and the meta-characters (,), + and ?.

Arguments, if present, must be grouped together in the second position on the command line. The character of the group must be a –. Unless the –f option is given, the next argument is always taken to be the expression. If –f is present then the third argument is the name of the file containing the expression.

Beware of spaces or tabs in the expression, even if your compiler supports quoted arguments. CP/M will object to ^I anywhere on the command line. Use \s for spaces and \t for tabs to be safe.

Some of the command line switches do mutually exclusive things (like –ef and –eh). If you try to trick grep into doing something it is not supposed to do, the output will be undefined.

Grep's execution speed varies as a function of the type of expression being parsed. The speed will vary as follows (listed fastest to slowest):

- Simple expressions anchored to beginning of line (^<expression>).
- Expressions matching literal strings.
- Expressions including character classes ([ ]).
- Expressions including closure (*).

Figure 1

(Figure 2, below). Matchs( ) is passed a pointer to this linked list. A string holding a regular expression is converted to a template by the procedure makepat( ). Since alloc( ) is used to allocate the (main) memory needed to contain the template, the expression can be any length, within reason. Using the routine unmakepat( ), you can return the memory used by a template to the free list. Unmakepat( ) is not used by grep, but it may be useful for other applications (such as editors).

Some of the fields in a TOKEN are not always used; although using a union would have saved a small amount of (main) memory space, this would have added additional complexity to the program as a whole. The tok field identifies the type of symbol represented by this node (closure, character class, etc.). If the node is a literal character, 1char holds the character itself, and if the node is a character class, string points at a string holding all the characters in the class. Classes defined with the dash notation (a – z) are expanded.

Note that a CLOSURE token is put into the chain in front of the node on which it operates (even though the closure symbol is put after the character in the expression itself). This transposition eliminates the need for any sort of look-ahead in the searching routines. When we encounter the CLOSURE token, we know the next token should be repeated zero or more times. If the CLOSURE token didn't come first, we would process the character on which the CLOSURE operates as if it were a literal match; that is, we would match a single occurrence of the character. Since closure represents zero or more matches, this first match would be incorrect. So, if the CLOSURE token didn't come first, we couldn't process a token without also having to look for a CLOSURE token following it.

Matchs( )

The core of grep is the routine matchs( ) and the procedures that matchs( ) calls. This routine looks for a regular expression match in a string. It takes the string and a pointer to a pattern template as input and returns a pointer into the string upon success (or zero if no match is found). This pointer can point at either the beginning or the end of the matched string, depending on the value of the "ret_endp" parameter to matchs( ). If ret_endp is zero, a pointer to the beginning of the matched string is returned; if ret_endp is non-zero, then a pointer to the end of the matched string is returned. For example, given the string "abcdefghijklm" and the pattern a.*j, matchs( ) can return a pointer either to the "a" or to the "j." This is a useful feature in an editor.

You must be careful with the $ symbol if you want to use the pointers returned by matchs( ). Usually $ means *at* the end of line, not the end-of-line character itself; for example, if you are searching for f$, a pointer will be returned to the "f." However, if you are searching for $ itself, a pointer to the actual end-of-line character is returned. This takes care of the ^$ case. Matchs( ) must return a pointer to something, and the only character on the line is the newline character. A search for \n will always return a pointer to the newline. This last is a nonstandard feature but, again, is useful in an editor application.

Matchs( ) advances through the input string, one character at a time, until it reaches the end of the string and failure is returned. It calls amatch( ) to actually do the comparisons; when amatch( ) returns success, so does matchs( ).

Amatch( ) goes through the pattern template, one element at a time, comparing it with the text string. It advances to the next element of the template with each successful comparison, also advancing the text string as appropriate. If amatch( ) reaches the end of the template, the match is successful. Omatch( ) is called to do the simple comparisons: single characters against single elements in the pattern template.

Amatch( ) returns immediately on failure so the performance of matchs( ) is not too slow in the general case (execution time is directly proportional to the length of the input stream). The worst-case performance, however, is an exponential function of the matched string's length. Given an input string of the form

aaaa

lower-case word ending in a "t"), the final "t" will be sucked up by the first scan since "t" is included in the character class [a – z]. Since amatch( ) has scanned too far, an attempt to match the "t" will now fail. So, it backs up a notch in the input string then tries to match the rest of the pattern template again, repeating this process recursively until it gets back to the beginning of the closure. The recursion only goes one level deep. If anyone knows a better way to do this, please tell me.

Examples

Makepat( ) takes two arguments. The first argument is an ASCII string holding the regular expression; the second argument is a character to use as a terminator in the expression string. That is, processing of the expression string will be terminated when the character specified by the second argument is encountered.

The template returned by makepat( ) when called with

makepat("^The qui", '\0')

is shown in Figure 3 (below). If we had called makepat with

makepat("^The quick", ' ')

a template like that in Figure 3 would be returned. However, this time the template would stop with the "e" LITCHAR because we passed makepat( ) a space as the input string terminator. A call to

makepat("a[0 – 9].*[^v]$",' \0')

returns a pointer to the template shown in Figure 4 (page 56).

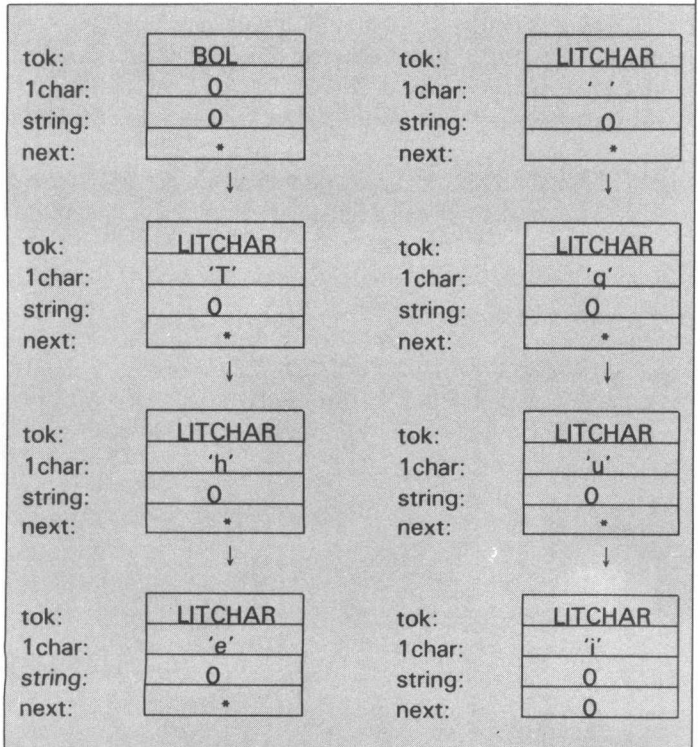


Figure 3

Matchs( ) looks for an expression on only one input line. Consequently, it has to be called several times, once for each line in the input file. Three arguments are required: the first is a pointer to the line being searched, the second is a pointer to a pattern template (returned by a previous call to makepat( ), and the third determines whether matchs( ) will return a pointer to the beginning or to the end of the matched pattern (0 for a pointer to the beginning, 1 for the end).

Consider the program fragment:

```
#include "tools.h"
TOKEN *template; char *ptr;
template = makepat("456", '\0');
ptr = matchs("1234567890", template, 0);
ptr = matchs("1234567890", template, 1);
ptr = matchs("abcdefghij", template, 1);
```

The call to makepat( ) returns a pointer to a pattern template representing the string "456." This template will be three elements long, one element for each character in the string, and all three elements will be of type LITCHAR. The first call to matchs( ) will return a pointer to the "4" (because its third parameter is zero). The second call to matchs( ) will return a pointer to the "6" (because the third parameter is 1). The third call to matchs( ) returns zero because the string "456" doesn't exist in the string "abcdefghij."

A simplistic version of grep—using only gets( ), makepat( ), and matchs( )—is shown in Figure 5 (page 57). This version prints all input lines that match a pattern found on the command line. No attempt at any sort of error checking is made in this example, so it's not a very practical program. It does illustrate how makepat( ) and matchs( ) may be used in a real program.

Debugging Aids

Four routines are included here for use in debugging. These are pr_tok( ), pr_line( ), insert( ), and delete( ); all are in tools.c.

Pr_tok( ), when passed a pointer to a linked list of TOKENs, will print out the list to stdout. You can use pr_tok( ) to monitor the progress of amatch( ) as it works and to see if the expression is translated correctly to begin with.

Pr_line( ) prints out one line of text to stdout; any nonprintable characters are represented as numbers in the form:

\0x(two hex digits)

Insert( ) puts a character into a string at a place pointed to by its "str" parameter. Delete( ) takes the character out again.

Some Implementation Notes

In the course of bringing up grep on my own system, I ran into a few problems worth mentioning. First, depending on which compiler you use, getting the expression from the command line may be unexpectedly difficult. The command-line parser for Aztec C II (the compiler I used on this version) doesn't allow quoted strings; that is, an argument of the form

grep "this is a single argument" foo.bar

will be broken up into seven (rather than three) arguments by the compiler. So I modified the command-line parser in the module croot.c to accept quoted strings.

While the BDS C compiler does give you quoted strings, pitfalls exist here too. As it parses the command line, BDS strips off the quotes. Because BDS wild card expansion is done with a call to wildexp( ) inside of the program proper (instead of inside the command-line parser where it belongs), wildexp( ) can't differentiate between the quoted argument and the normal arguments: it doesn't have any quotes to work with. Consequently, it will try to expand the regular expression if the expression has an * or a ? in it.

I got around this problem in a BDS version of grep by getting the regular expression from the command line *before* calling wildexp( ). I then replaced the argv entry, which pointed at the expression, with a pointer to a null string and called wildexp( ). You could also try to call wildexp( ) from inside the command-line parser itself. Since the parser is written in assembly language and wildexp( ) is in C, I didn't try this (though it would be a permanent solution to the problem).

A similar problem can be found in microshell. Microshell supports quoted strings, but a backslash inside a quoted string is treated as a special character. You need to double the backslash to pass it through to grep (i.e., use \\s instead of \s; and \\\\ instead of \\). One saving grace is that microshell lets you pass tabs through unmolested.

One final difficulty. You may use grep as a filter if you like: if you are running microshell or some other environment that supports pipes, you may use grep as a general purpose filter, stripping out unwanted material from the input stream and passing the modified stream on to another program. The problem is what happens to end-of-line terminators on their way through the pipe.

CP/M requires a carriage return, line feed combination at the end of line. C, however, wants a single newline character (\n). Consequently, when getc( ) sees a CR, it echoes it as a CR–LF (so the screen looks nice) and then turns it into a \n; on output, putc( ) will turn the \n back into a CR–LF.

This is fine until you use the output of one program as the input of another. The next program will see the CR–LF, echo it as CR–LF–LF, and map it to a \n\n (one \n for the CR, another for the LF). The output from the second program will have a CR–LF–CR–LF at the end of every line: instant double spacing. If you go through another layer of pipe you will get CR–LF–CR–LF–CR–LF–CR–LF at the end of each line, and so on.

A solution would be to have getchar( ) work as described above and have getc( ) ignore the LF character entirely (not pass it through to the program). The BDS C compiler doesn't lend itself to this change because its I/O library has the two input routines functionally reversed (i.e., getc( ) calls getchar( ), which is backwards from Unix). You could also use the BDS version 1.5 raw I/O routines, but then your code would be even more nonstandard. Alternately, you could do all your character input from the console with direct bdos( ) calls.

Conclusion

In spite of the few implementation problems I encountered, grep remains an extremely useful program. It has saved me hours of rooting around in modular C programs looking for misspelled subroutine names. Its cross-referencing capability has also proved invaluable. When I get a new C compiler, the first thing I do is use grep to make a cross reference of the

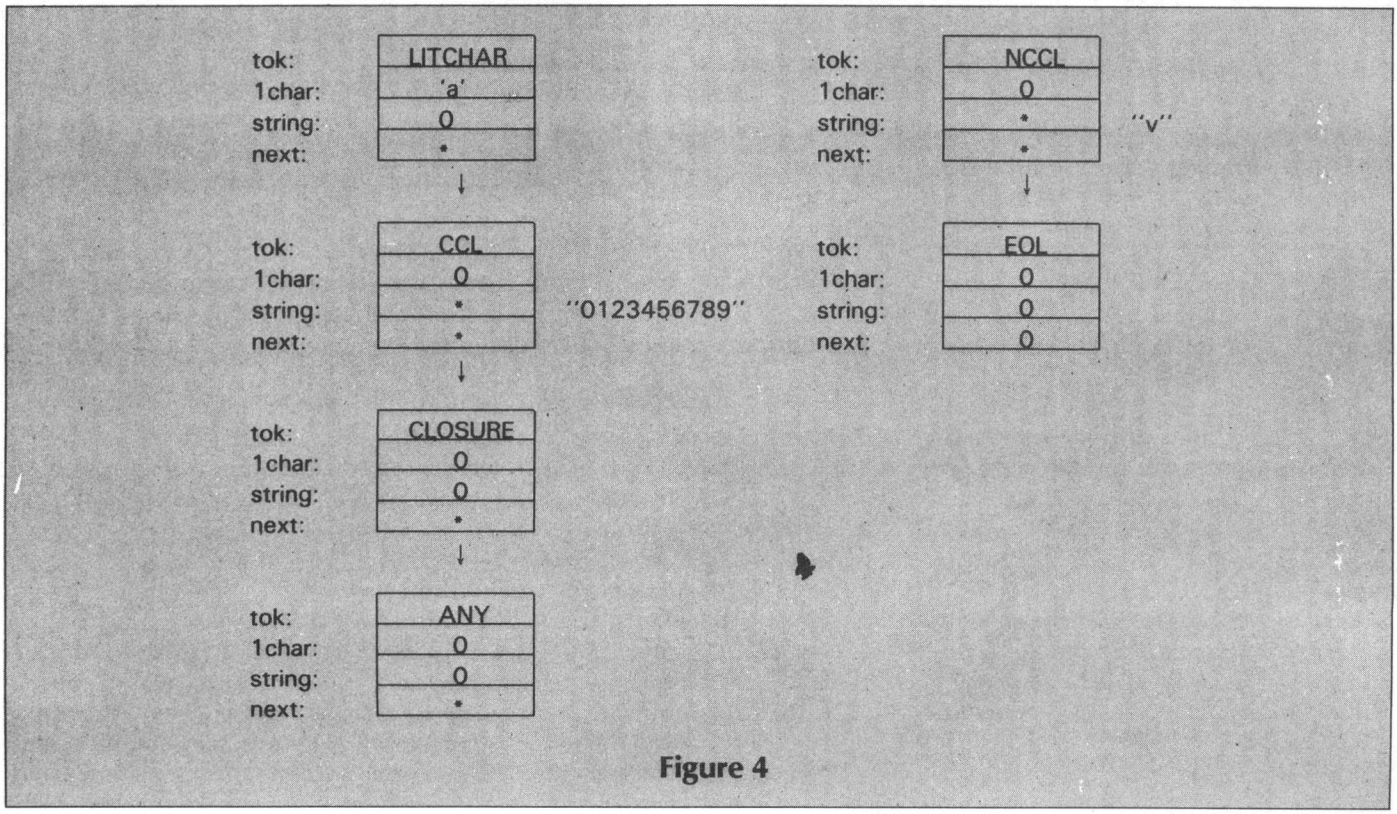


Figure 4

```
#include "stdio.h"
#include "tools.h"

main(argc, argv)
int    argc;
char  *argv[ ];
{M8•X
        TOKEN  *template;
        char    str[132];

        template = makepat( argv[1], '\0' );

        while(  gets( str ) != NULL )
                if( matches( str, template, 0 ) )
                    printf("%s\n", str );
}
```

Figure 5

runtime library sources. Using this cross reference, it's easy to find the source code for the particular library subroutine that doesn't seem to be working correctly . The addition of matchs( ) to the RED editor has made it a much nicer editor, giving RED not only an extended search capability but also a powerful global substitution capability. Once you've used regular expressions in an editor, you won't settle for anything else. I hope that you find this program as useful as I have.

Copies of grep, along with the C source code, are available from the author at the above address for $35.00 (+ tax if a California resident). (Copies provided on a standard, IBM format, single density, 8-inch, CP/M-compatible floppy disk or on DS/DD, IBM-PC, PCDOS-Compatible, 5¼-inch disks). **DDJ**

GREP.C (Listing Continued, text begins on page 50)

Listing One

```
/*---------------------------------------------------------------
 *         TOOLS.H: Various #defines and typedefs for GREP
 *
 *                Copyright (c) 1984 Allen Holub
 *      Copyright (c) 1984 Software Engineering Consultants
 *                        P.O. Box 5679
 *                    Berkeley, CA,  94705
 *
 *                    All rights reserved.
 *
 *      This program may be copied for personal,  non-commercial use
 *      only, provided that this copyright notice is included in all
 *      copies  and  that  this  program is not modified in any way.
 *      Copying for any other use without previously  obtaining  the
 *      written permission of the author is prohibited.
 *
 *      Machine readable versions of  this  program may be purchased
 *      for $35 from  Software  Engineering  Consultants.  Supported
 *      disk formats are CP/M 8" SS/SD and PCDOS (v2.x) 5-1/4" DS/DD.
 *
 *---------------------------------------------------------------
 */

/*
 *      #defines for non-printing ASCII characters
 */

#define NUL     0x00    /* ^@   */
#define SOH     0x01    /* ^A   */
#define STX     0x02    /* ^B   */
#define ETX     0x03    /* ^C   */
#define EOT     0x04    /* ^D   */
#define ENQ     0x05    /* ^E   */
#define ACK     0x06    /* ^F   */
#define BEL     0x07    /* ^G   */
#define BS      0x08    /* ^H   */
#define HT      0x09    /* ^I   */
#define LF      0x0a    /* ^J   */
#define NL      LF
#define VT      0x0b    /* ^K   */
#define FF      0x0c    /* ^L   */
#define CR      0x0d    /* ^M   */
```

```
#define SO      0x0e    /* ^N   */
#define SI      0x0f    /* ^O   */
#define DLE     0x10    /* ^P   */
#define DC1     0x11    /* ^Q   */
#define DC2     0x12    /* ^R   */
#define DC3     0x13    /* ^S   */
#define DC4     0x14    /* ^T   */
#define NAK     0x15    /* ^U   */
#define SYN     0x16    /* ^V   */
#define ETB     0x17    /* ^W   */
#define CAN     0x18    /* ^X   */
#define EM      0x19    /* ^Y   */
#define SUB     0x1a    /* ^Z   */
#define CPMEOF  SUB
#define ESC     0x1b    /* ^[   */
#define FS      0x1c    /* ^\   */
#define GS      0x1d    /* ^]   */
#define RS      0x1e    /* ^^   */
#define US      0x1f    /* ^_   */
#define DEL     0x7f    /* DEL  */


#define TRUE    1
#define FALSE   0


/*      Definitions of meta-characters used in pattern matching routines.
 *      LITCHAR & NCCL are only used as token identifiers; all the others
 *      are also both token identifiers and the actual symbol used in
 *      the regular expression
 */


#define BOL     '^'
#define EOL     '$'
#define ANY     '.'
#define LITCHAR 'L'
#define ESCAPE  '\\'
#define CCL     '['             /* Character class:     [...]          */
#define CCLEND  ']'
#define NEGATE  '^'
#define NCCL    '!'             /* Negative character class [^...]     */
#define CLOSURE '*'
#define OR_SYM  '|'

#define CLS_SIZE  128           /* Largest permitted size for an expanded
                                ** character class.  (Ie. the class [a-z]
                                ** will expand into 26 symbols; [a-z0-9] will
                                ** expand into 36 symbols.)
                                */

/*
 *      Tokens are used to hold pattern templates. (see makepat() in

 *      tools.h
 */

typedef struct token{
                char            tok;
                char            lchar;
                char            *string;
                struct token    *next;
              }TOKEN;

#define TOKSIZE sizeof(TOKEN)
```

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)

Listing One

```
/*
 *      An absolute maximum for strings.
 */

#define MAXSTR          132             /* Maximum number of characters in
                                         *      a line.
                                         */

extern  char    *matchs();
extern  int     amatch();
extern  char    *in_string();
extern  TOKEN   *getpat();
extern  int     esc();
extern  int     dodash();
extern  TOKEN   *makepat();
extern  int     unmakepat();
extern  int     insert();
extern  int     delete();
extern  int     isalphanum();
extern  int     stoupper();
extern  int     pr_tok();
extern  int     pr_line();
extern  int     max();
```

End Listing One

Listing Two

```
/*---------------------------------------------------------------
 *         TOOLS.C: The expression parser used by grep.
 *
 *                 Copyright (c) 1984 Allen Holub
 *      Copyright (c) 1984 Software Engineering Consultants
 *                          P.O. Box 5679
 *                      Berkeley, CA,  94705
 *
 *                      All rights reserved.
 *
 *      This program may be copied for personal,  non-commercial use
 *      only, provided that this copyright notice is included in all
 *      copies  and  that  this  program is not modified in any way.
 *      Copying for any other use without previously  obtaining  the
 *      written permission of the author is prohibited.
 *
 *      Machine readable versions of  this  program may be purchased
 *      for $35 from  Software  Engineering  Consultants.  Supported
 *      disk formats are CP/M 8" SS/SD and PCDOS (v2.x) 5-1/4" DS/DD.
 *
 *---------------------------------------------------------------
 */

#include "a:stdio.h"
#include "b:tools.h"

/*
 *      This module contains the various routines needed by grep
 *      to match regular expressions. Routines are ordered
 *      alphabeticaly.
 */

int     amatch( lin, pat, boln )
char    *lin, *boln;
TOKEN   *pat;
{
```

```
        /*    Scans through the pattern template looking for a match
         * with lin. Each element of lin is compared with the template
         * until either a mis-match is found or the end of the template
         * is reached. In the former case a 0 is returned; in the latter,
         * a pointer into lin (pointing to the last character in the
         * matched pattern) is returned.
         *
         *      "lin"   is a pointer to the line being searched.
         *      "pat"   is a pointer to a template made by makepat().
         *      "boln"  is a pointer into "lin" which points at the
         *                      character at the beginning of line.
         */

register char   *bocl, *rval, *strstart;


if (pat == 0)
        return (0);

strstart = lin;

while ( pat )
{
        if (pat->tok == CLOSURE  &&   pat->next)
        {
                /*
                 *      Process a closure:
                 *      First skip over the closure token to the
                 *      object to be repeated. This object can be
                 *      a character class.
                 */

                pat = pat->next;

                /*      Now match as many occurrences of the
                 *      closure pattern as possible.
                 */

                bocl = lin;

                while ( *lin  &&  omatch(&lin, pat)  )
                        ;


                /*     'Lin' now points to the character that made
                 *      made us fail. Now go on to process the
                 *      rest of the string. A problem here is
                 *      a character following the closure which
                 *      could have been in the closure.
                 *      For example, in the pattern "[a-z]*t" (which
                 *      matches any lower-case word ending in a t),
                 *      the final 't' will be sucked up in the while
                 *      loop. So, if the match fails, we back up a
                 *      notch and try to match the rest of the
                 *      string again, repeating this process
                 *      recursively until we get back to the
                 *      beginning of the closure. The recursion
                 *      goes, at most, two levels deep.
                 */

                if (pat = pat->next)
                {
                        while ( bocl <= lin )
                        {
                                if (rval = amatch(lin, pat, boln) )
                                {
                                        /* success */
```

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)

Listing Two

```
                                        return(rval);
                                }
                                else
                                        --lin;
                        }
                        return (0);     /* match failed */
                        }
                }
                else if ( omatch(&lin, pat, boln) )
                {
                        pat = pat->next;
                }
                else
                {
                        return (0);
                }
        }

        /*
         *      Note that omatch() advances lin to point at the next
         *      character to be matched. Consequently, when we reach
         *      the end of the template, lin will be pointing at the
         *      character following the last character matched.
         *      The exceptions are templates containing only a
         *      BOLN or EOLN token. In these cases omatch doesn't
         *      advance.
         *
         *      So, decrement lin to make it point at the end of the
         *      matched string. Then, check to make sure that we haven't
         *      decremented past the beginning of the string.
         *
         *      A philosophical point should be mentioned here. Is $
         *      a position or a character? (Ie. does $ mean the EOL
         *      character itself or does it mean the character at the end of
         *      the line.) I decided here to make it mean the former, in
         *      order to make the behavior of amatch() consistent. If you
         *      give amatch the pattern ^$ (match all lines consisting only
         *      of an end of line) then, since something has to be returned,
         *      a pointer to the end of line character itself is returned.
         *
         *      One final point. If you use a macro instead of a real
         *      subroutine to define max(), then take the --lin out of
         *      the macro call to avoid side-effects (lin being decremented
         *      twice).
         */

        return ( max(strstart , --lin) );
}

/* ---------------------------------------------------------------------- */

delete( ch, str )
int             ch;
register char   *str;
{
        /*      Delete the first occurrence of character from string
         *      moving everything else over a notch to fill the hole.
         */

        ch &= 0xff;

        while ( *str && *str != ch)
                str++;

        while ( *str )
        {
```

```
                *str = *(str+1);
                str++;
        }
}

/* ---------------------------------------------------------------------- */

int     dodash(delim, src, dest, maxccl)

int     delim, maxccl;
char    **src, *dest;
{
        /*      Expand the set pointed to by *src into dest.
         *      Stop at delim. Return 0 on error or
         *      size of character class on success. Update *src to point
         *      at delim. A set can have one element {x} or several
         *      elements ( {abcdefghijklmnopqrstuvwxyz} and {a-z}
         *      are equivalent ). Note that the dash notation is expanded
         *      as sequential numbers. This means (since we are using the
         *      ASCII character set) that a-Z will contain the entire alphabet
         *      plus the symbols:  [\]^_`  . The maximum number of characters
         *      in a character class is defined by maxccl.
         */

        register char   *dstart;
        register int    k, at_begin;
        char            *sptr;

        dstart = dest;
        sptr = *src;
        at_begin = 1;

        while ( *sptr  &&  (*sptr != delim)   &&   (dstart-dest < maxccl)  )
        {
                if ( *sptr == ESCAPE )
                {
                        *dest++ = esc(&sptr);
                        sptr++;
                }

                else if ( *sptr != '-')
                        *dest++ = *sptr++;

                else if ( at_begin ||  *(sptr+1) == delim )
                        *dest++ = '-';                      /* literal '-' */

                else if ( *(sptr -1 ) <= *(sptr+1) )
                {
                        sptr++;

                        for( k= *(sptr-2) ; ++k <= *sptr ;)
                                *dest++ = k;

                        sptr++;
                }
                else
                {
                        return(0);
                }

                at_begin = 0;
        }

        *dest++ = '\000' ;
        *src = sptr;

        return (dest - dstart);
}

/* ---------------------------------------------------------------------- */

int     esc(s)
char    **s;
```

(Continued on next page)

```
{
        register int    rval;

        /* Map escape sequences into their equivalent symbols. Returns the
         * Correct ASCII character. If no escape prefix is present
         * then s is untouched and  *s is returned, otherwise **s
         * is advanced to point at the escaped character and the
         * translated character is returned.
         */

        if ( **s != ESCAPE )
        {
                rval = **s;
        }
        else
        {
                (*s)++;

                switch( toupper(**s) )
                {
                case '\000':    rval = ESCAPE;          break;
                case 'S':       rval = ' '  ;           break;
                case 'N':       rval = '\n' ;           break;
                case 'T':       rval = '\t' ;           break;
                case 'B':       rval = '\b' ;           break;
                case 'R':       rval = '\r' ;           break;
                default:        rval = **s  ;           break;
                }
        }

        return (rval);
}

/* ---------------------------------------------------------------------- */

TOKEN   *getpat( arg )
char    *arg;
{
        /*      Translate arg into a TOKEN string
         */

        return ( makepat(arg, '\000' ) );
}

/* ---------------------------------------------------------------------- */

insert( ch, str )
int             ch;
register char   *str;
{
        /*      Insert ch into str at the place pointed to by str. Move
         *      everything else over a notch
         */

        register char   *bp;

        bp = str;

        while (*str)                    /* Find the end of string       */
                str++;
        do                              /* Move the tail over one notch */
        {
                *(str+1) = *str;
                str--;

        } while (str >= bp);
```

```
        *bp = ch;                       /* Put the char in the hole.    */
}

/* ------------------------------------------------------------------------ */

char    *in_string( delim, str )
register int    delim;
register char   *str;
{
        /*
         *  Return a pointer to delim if it is in the string, 0 if it is not.
         */

        delim &= 0x7f;

        while (*str  &&  *str != delim)
                str++;

        return ( *str ? str : 0 );
}

/* ------------------------------------------------------------------------ */

int isalphanum(c)
int     c;
{

        /*   Return true if c is an alphabetic character or digit,
         *      false otherwise.
         */

        return( ('a' <= c  &&  c <= 'z') ||
                ('A' <= c  &&  c <= 'Z') ||
                ('0' <= c  &&  c <= '9')
              );
}

/*------------------------------------------------------------------------*/

TOKEN   *makepat(arg, delim)
char    *arg;
int     delim;
{
        /*      Make a pattern template from the string pointed to by arg.
         *      Stop when delim or '\000' or '\n' is found in arg.
         *      Return a pointer to the pattern template.
         *
         *      The pattern templates used here are somewhat different
         *      than those used in the book; each token is a structure
         *      of the form TOKEN (see tools.h). A token consists of
         *      an identifier, a pointer to a string, a literal
         *      character and a pointer to another token. This last is 0 if
         *      there is no subsequent token.
         *
         *      The one strangeness here is caused (again) by CLOSURE which
         *      has to be put in front of the previous token. To make this
         *      insertion a little easier, the 'next' field of the last
         *      token in the chain (the one pointed to by 'tail') is made
         *      to point at the previous node. When we are finished,
         *      tail->next is set to 0.
         */

        TOKEN   *head, *tail;
        TOKEN   *ntok;
        char    buf[CLS_SIZE];
        int     error;

        /*      Check for characters that aren't legal at the beginning
         *      of a template.
         */

        if (*arg=='\0' ||  *arg==delim  ||  *arg=='\n' || *arg==CLOSURE)
                return(0);
```

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)

Listing Two

```
error = 0;
head = 0;
tail = 0;

while ( *arg  &&  *arg != delim  &&   *arg != '\n'  &&  !error)
{
        ntok = alloc( TOKSIZE );
        ntok->string = &(ntok->lchar);
        ntok->lchar = '\000';
        ntok->next = 0;

        switch(*arg)
        {
        case ANY:
                ntok->tok = ANY;
                break;

        case BOL:

                if (head==0)    /* then this is the first symbol */

                        ntok->tok = BOL;
                else
                        error = 1;
                break;

        case EOL:

                if ( *(arg+1) == delim   || *(arg+1) == '\000'
                                         || *(arg+1) == '\n'  )
                        ntok->tok = EOL;
                else
                        error = 1;
                break;

        case CCL:

                if (*(arg+1) == NEGATE)
                {
                        ntok->tok = NCCL;
                        arg += 2;
                }
                else
                {
                        ntok->tok = CCL;
                        arg++;
                }

                error = dodash(CCLEND, &arg, buf, CLS_SIZE) ;
                if (error != 0)
                {
                        ntok->string = alloc( error );
                        strcpy( ntok->string, buf );
                        error = 0;
                }

                break;

        case CLOSURE:

                if ( head != 0 )
                {
                        switch ( tail->tok )
                        {
                        case BOL:
                        case EOL:
```

```
                        case CLOSURE:
                                return(0);

                        default:
                                ntok->tok = CLOSURE;
                        }
                }
                break;

        default:
                ntok->tok = LITCHAR;
                ntok->lchar = esc(&arg);
        }

        if( error  ||  ntok == 0 )
        {
                unmakepat(head);
                return (0);
        }

        else  if (head == 0)
        {
                /*  This is the first node in the chain.
                 */

                ntok->next = 0;
                head = tail = ntok;
        }

        else if (ntok->tok != CLOSURE)

        {
                /* Insert at end of list (after tail) */

                tail->next = ntok;
                ntok->next = tail;
                tail = ntok;
        }

        else if(head != tail)
        {
                /* More than one node in the chain. Insert the
                 * CLOSURE node immediately in front of tail.
                 */

                (tail->next)->next = ntok;
                ntok->next = tail;
        }

        else
        {
                /* Only one node in the chain, Insert the CLOSURE
                 * node at the head of the linked list.
                 */

                ntok->next = head;
                tail->next = ntok;
                head = ntok;
        }

        arg++;
}

tail->next = 0;
return (head);
```

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)

Listing Two

```
}

/* ---------------------------------------------------------------------- */

char    *matchs(line, pat, ret_endp)
char    *line;
TOKEN   *pat;
int     ret_endp;
{
        /*
         *      Compares line and pattern. Line is a character string while
         *      pat is a pattern template made by getpat().
         *      Returns:
         *              1. A zero if no match was found.
         *              2. A pointer the last character
         *                 satisfying the match if ret_endp is non-zero.
         *              3. A pointer to the beginning of the matched string
         *                 if ret_endp is 0;
         *
         *      For example:
         *
         *              matchs ("1234567890", getpat("4[0-9]*7"), 0);
         *
         *      will return a pointer to the '4', while
         *
         *              matchs ("1234567890", getpat("4[0-9]*7"), 1);
         *
         *      will return a pointer to the '7'.
         */

        char    *rval, *bptr;

        bptr = line;

        while (*line)
        {
                if ( (rval = amatch(line, pat, bptr)) == 0 )
                {
                        line++;
                }
                else
                {
                        rval = ret_endp ? rval : line ;
                        break;
                }
        }

        return (rval);
}

/* ---------------------------------------------------------------------- */

stoupper(str)
char    *str;
{
        /*
         *      Map the entire string pointed to by str to upper case
         *      Return str.
         */

        char    *rval;


        rval = str;

        while (*str)
```

```
        {
                if ( 'a' <= *str  &&  *str <= 'z' )
                        *str -= ('a' - 'A');

                str++;
        }

        return(rval);
}

/* ---------------------------------------------------------------------- */

int     max(x,y)
int     x,y;
{
        return ( (x>y) ?  x  :  y );
}

/* ---------------------------------------------------------------------- */

int     omatch (linp, pat, boln)
char    **linp, *boln;
TOKEN   *pat;
{
        /*      Match one pattern element, pointed at by pat, with the
         *      character at **linp. Return non-zero on match.
         *      Otherwise, return 0. *Linp is advanced to skip over the
         *      matched character; it is not advanced on failure. The
         *      amount of the advance is 0 for patterns that match null
         *      strings, 1 otherwise. "boln" should point at the position
         *      that will match a BOL token.
         */



        register int    advance;

        advance = -1;

        if ( **linp )
        {
                switch ( pat->tok )
                {
                case LITCHAR:
                        if ( **linp == pat->lchar )
                                advance = 1;
                        break;


                case BOL:
                        if ( *linp == boln )
                                advance = 0;
                        break;

                case ANY:
                        if ( **linp != '\n' )
                                advance = 1;
                        break;

                case EOL:
                        if ( **linp == '\n' )
                                advance = 0;
                        break;

                case CCL:
                        if( in_string (**linp, pat->string) )
                                advance = 1;
                        break;

                case NCCL:
                        if ( ! in_string (**linp, pat->string) )
                                advance = 1;
                        break;
```

(Continued on next page)

```
                default:
                        printf("omatch: can't happen\n");
                }
        }
        if (advance >= 0)
                *linp += advance;

        return( ++advance );
}
/* ---------------------------------------------------------------------- */

pr_line(ln)
register char   *ln;
{
        /*      Print out ln, if a non-printing character is found, print
         *      out its numerical value in the form "\0x<hex number>".
         *      Again, this is a debugging aid. It lets you see what's
         *      really on the line.
         */

        for ( ; *ln ; ln++ )


        {
                if ( (' ' <= *ln) && (*ln <= '~') )
                        putchar(*ln);
                else
                {
                        printf("\\0x%02x", *ln);

                        if (*ln == '\n')
                                putchar('\n');
                }
        }
}

/* ---------------------------------------------------------------------- */

pr_tok(head)
TOKEN   *head;
{
        register char   *str;

        /*      Print out the pattern template (linked list of TOKENs)
         *      pointed to by head. This is a useful debugging aid. Note
         *      that pr_tok() just scans along the linked list, terminating
         *      on a null pointer; so, you can't use pr_tok from inside
         *      makepat() because tail->next points to the previous
         *      node instead of being null.
         */

        for (; head ; head = head->next )
        {
                switch (head->tok)
                {
                case BOL:
                        str = "BOL";
                        break;

                case EOL:
                        str = "EOL";
                        break;

                case ANY:
                        str = "ANY";
                        break;

                case LITCHAR:
                        str = "LITCHAR";
                        break;
```

```
                case ESCAPE:
                        str = "ESCAPE";
                        break;

                case CCL:
                        str = "CCL";
                        break;

                case CCLEND:
                        str = "CCLEND";
                        break;

                case NEGATE:
                        str = "NEGATE";
                        break;

                case NCCL:
                        str = "NCCL";
                        break;

                case CLOSURE:
                        str = "CLOSURE";
                        break;

                default:
                        str = "**** unknown ****";
                }

                printf("%-8s at: 0x%x, ", str, head);

                if (head->tok == CCL || head->tok == NCCL)
                        printf ("string =[%s]=, ", head->string );

                else if (head->tok == LITCHAR)
                        printf("lchar = %c, ", head->lchar);

                printf("next = 0x%x\n", head->next);
        }

        putchar('\n');
}

/* ---------------------------------------------------------------- */

unmakepat(head)
TOKEN   *head;
{
        /*      Free up the memory used for the token string    */
        register TOKEN  *old_head;

        while (head)
        {
                switch (head->tok)
                {
                case CCL:
                case NCCL:
                        free(head->string);
                        /* no break, fall through to default */

                default:
                        old_head = head;
                        head = head->next;
                        free(old_head);
                        break;
                }
        }
}
```

End Listing Two

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)

Listing Three

```
/*-------------------------------------------------------------
 *        GREP.C: A generalized regular expression parser.
 *
 *                Copyright (c) 1984 Allen Holub
 *      Copyright (c) 1984 Software Engineering Consultants
 *                        P.O. Box 5679
 *                    Berkeley, CA,  94705
 *
 *                    All rights reserved.
 *
 *      This program may be copied for personal,  non-commercial use
 *      only, provided that this copyright notice is included in all
 *      copies  and  that  this  program is not modified in any way.
 *      Copying for any other use without previously  obtaining  the
 *      written permission of the author is prohibited.
 *
 *      Machine readable versions of  this  program may be purchased
 *      for $35 from  Software  Engineering  Consultants.  Supported
 *      disk formats are CP/M 8" SS/SD and PCDOS (v2.x) 5-1/4" DS/DD.
 *
 *-------------------------------------------------------------
 */

#include "a:stdio.h"
#include "b:tools.h"

/*
 *      GREP
 *
 *      Search a file for a pattern.
 *
 *      The algorithm used here is essentially the algorithm in
 *      Software Tools in Pascal (pp 145f.). Though the routines have
 *      been changed somewhat to put them into good 'C'. See tools.c
 *      for details.
 *
 *      This program is a healthy subset of the UNIX program of the same
 *      name. The differences are as follows:
 *
 *              - the -s, -x and -b options are not supported.
 *              - the meta-characters ()+? are not supported.
 *
 *      usage is:
 *              grep [-vclnhyef] [expression] files ...
 *
 *      To compile with the Computer Inovations C86 compiler use:
 *
 *              cc1 grep
 *              cc2 grep
 *              cc3 grep
 *              cc4 grep
 *              cc1 tools
 *              cc2 tools
 *              cc3 tools
 *              cc4 tools
 *              link grep tools,,,c86s2s.lib
 *
 *      To compile with Aztec CII use:
 *
 *              cc1 -x4000 grep.c
 *              as  grep.asm
 *              cc1 -x4000 tools.c
 *              as  tools.asm
 *              ln  grep.o tools.o a:libc.lib
 */
```

```
#define CPM     1               /*      Comment this out if you're compiling
                                 *      in an MSDOS system.
                                 */

#define MAXLINE  128            /*      Maximum size of an input line
                                 */

#define MAX_EXPR 64             /*      The maximum number of regular
                                 *      expressions separated by
                                 *      newlines or | allowed.
                                 */


/*      The following global flags are true if a switch was set
 *      in the command line, false otherwise.
 */

int     vflag, yflag, cflag, lflag, nflag, hflag, fflag;

main(argc,argv)
int     argc;
char    **argv;
{
        int     i, j, linenum, count;

        int     line[MAXLINE];
        int     numfiles;
        FILE    *stream;
        int     exprc;


        TOKEN   *exprv[MAX_EXPR];

        i = 1;

        if (argc < 2)
                abort( pr_usage(1) );

        if ( *argv[i] == '-')
        {
                /*
                 *      Expand the switches on the command line
                 */

                expand_sw( argv[i++] );

                if ( i == argc )
                        abort( pr_usage(1) );
        }

        /*      Get the pattern string.
         */

        if ( (exprc = get_expr( exprv, MAX_EXPR, &argv[i++])) == 0 )
                abort( pr_usage(2) );

        numfiles = argc - i;            /*  Get number of files left to
                                         *  process on the command line
                                         */

        fprintf(stderr,"(c) Copyright 1984, Software Engineering Consultants\n"
        do
        {
                if ( numfiles)
                {
                        stream = fopen( argv[i], "r");
```

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)

Listing Three

```
                        if (stream == NULL)
                        {
                                fprintf(stderr, "Can't open %s\n", argv[i]);
                                continue;
                        }
                }
                else
                {
                        stream = stdin;
                }

                count = 0;
                linenum = 1;

                while ( fgets(line, MAXLINE, stream) )
                {
#ifdef CPM
                        if (!fflag  ||  yflag )
                                stoupper(line);
#else
                        if ( yflag )
                                stoupper(line);
#endif

                        for( j = exprc ; --j >= 0 ; )
                        {
                                if ( matchs(line , exprv[j]) )
                                {
                                        count++;
                                        pr_match(linenum, line, argv[i], 1,
                                                                numfiles);
                                }
                                else
                                {
                                        pr_match(linenum, line, argv[i], 0,
                                                                numfiles);
                                }

                                linenum++;
                                cntrl_c();
                        }
                        if( lflag && count )
                                break;
                }
                pr_count( numfiles, argv[i], count );
                fclose (stream);

        } while (++i < argc);

        abort();
}

/*------------------------------------------------------------------------*/


pr_count( fcount, fname, count)
int     fcount, count;
char    *fname;
{
        /*      Process the -c flag by printing out a count and,
         *      if more than one file was listed on the command line,
         *      the file name too.
         */
```

```
        if (!cflag)
                return;

        if (fcount > 1)
                 printf("%-12s: ", fname );

        printf( "%d\n", count );
}

/*---------------------------------------------------------------------*/


pr_match(linenum, line, fname, match, numfiles)
int     linenum, match;
char    *line, *fname;
{
        /*      If a match is found print the correct thing
         *      as specified by the command line switches.
         */

        char    buf[80];

        if (cflag)
                return;

        if ( (vflag && !match) ||  (!vflag && match) )
        {
                if (!hflag && ( (numfiles > 1) || lflag) )
                        printf("%s%s", fname, lflag ? "\n" : ":" );

                if (nflag)
                        printf("%03d:", linenum );

                if (!lflag)
                        printf("%s", line );
        }
}

/*---------------------------------------------------------------------*/

pr_usage(num)
int     num;
{


#ifdef DEBUG
        fprintf(stderr,"%d ", num);
#endif
        fprintf(stderr,"usage: grep [-cefhlnvy] [expression] <files ...>\n");
}


/*---------------------------------------------------------------------*/


abort()
{
        exit();
}


/*---------------------------------------------------------------------*/


expand_sw( str )
char    *str;
{
        /*      Set global flags corresponding to specific switches
         *      if those switches are set
         */

        vflag = 0;
        cflag = 0;
```

(Continued on next page)

GREP.C (Listing Continued, text begins on page 50)

Listing Three

```
        lflag = 0;
        nflag = 0;
        hflag = 0;
        fflag = 0;
        yflag = 0;

        while (*str)
        {
                switch ( toupper(*str))
                {
                case '-':
                case 'E':
                        break;

                case 'C':
                        cflag = 1;
                        break;

                case 'F':
                        fflag = 1;

                        break;

                case 'H':
                        hflag = 1;
                        break;

                case 'L':
                        lflag = 1;
                        break;

                case 'N':
                        nflag = 1;
                        break;
                case 'V':
                        vflag = 1;
                        break;

                case 'Y':
                        yflag = 1;
                        break;

                default:
                        pr_usage(3);
                        abort();
                        break;
                }

                str++;
        }
}

/*---------------------------------------------------------------------*/

int do_or( lp, expr, max )
char    *lp;
TOKEN   **expr;
int     max;
{
        int     found;
        TOKEN   *pat;
        char    *op;

        found = 0;

        /*
         *      Extract regular expressions separated by OR_SYMs from
```

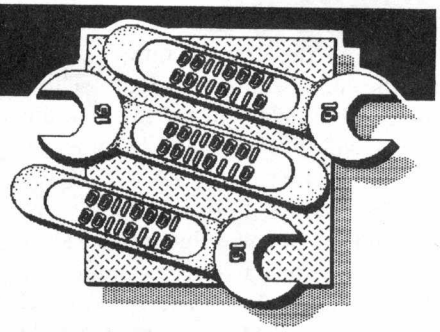
```
         *      lp and put them into expr. Extract only up to
         *      max expressions. If yflag is true map string to upper
         *      case first.
         */

        if( yflag )
                stoupper( lp );

        while ( op = in_string(OR_SYM, lp) )
        {
                if(found <= max && (pat = makepat(lp, OR_SYM)) )
                {
                        *expr++ = pat;
                        found++;
                }
                lp = ++op;

                if ( pat == 0 )
                        goto fatal_err;
        }

        if (found <= max  &&  (pat = makepat( lp, OR_SYM))  )
        {
                found++;
                *expr = pat;
        }


        if ( pat == 0 )
        {
fatal_err:
                printf("Illegal expression\n");
                exit();
        }

        return (found);
}

/*-------------------------------------------------------------------------*/


get_expr( expr, max,  defexpr )
TOKEN   *expr[];
int     max;
char    **defexpr;
{
        FILE    *stream;
        int     count;
        char    line[MAXLINE];

#ifdef DEBUG
        int     i;
#endif

        /*      Get regular expressions separated by | or newlines
         *      either out of a file or off the command line depending
         *      on whether the -f flag is set. The expressions are
         *      converted into pattern templates (see tools.c) and
         *      pointers to the templates are put into the array expr[]
         *      (which works similar to argv).
         *
         *      Return the number of expressions found (which can be used
         *      in a similar fashion to argc).
         */
```

GREP.C (Listing Continued, text begins on page 50)

Listing Three

```
count = 0;

if ( fflag )
{
        /*
         *      Then *defexpr is the file name and expressions should
         *      be taken from that file.
         */

        if ( (stream = fopen(*defexpr, "r")) == NULL )
        {
                fprintf(stderr, "Can't open %s\n", *defexpr);
                abort();
        }

        while ( (max - count)  &&  fgets(line, MAXLINE, stream) )
        {
                count += do_or(line, &expr[count], max - count );
        }

        fclose (stream);
}
else
{
        /*
         *      *defexpr is the expression itself.
         */

        if ( count += do_or( *defexpr, &expr[count], max - count))
                *defexpr = " ";
}

#ifdef DEBUG

        /*      Print out all the regular expressions after they have been
         *      converted into pattern templates (see tools.c).
         */

        for (i = count; --i >= 0 ; )
        {
                pr_tok(expr[i]);
                printf("------------------------------------------------\n");
        }

#endif
        return(count);
}


/*---------------------------------------------------------------------*/

cntrl_c()
{

#ifdef CPM

        /*      If any character was hit, and that character is a
         *      ^C, then abort.
         */

        if ( bdos(11) && ((bdos(1,0) & 0x7f) == 0x03) )
                abort();
#endif

}
```

End Listings



by Ray Duncan

MSDOS 2.0 Filters

One of the enhancements in MSDOS 2.0 is support for the concept of redirectable I/O á la Unix. Two predefined I/O channels, called the "standard input" and "standard output," may be accessed by any program; ordinarily they are directed to the keyboard and video display, respectively, but they can be individually redirected to other devices or to files with parameters placed in a DOS command line.

The standard I/O devices allow introduction of "filters," another feature of Unix, into the MSDOS environment. To quote the MSDOS manual, "A filter is a program or command that reads data from a standard input device, modifies the data, and writes the result to a standard output device . . . . By using the piping feature, you can cause a filter to receive its input from another command, or send its output to another command."

Three simple filters are supplied with MSDOS 2.0: SORT, which sorts text data; FIND, which searches an input stream to match a specified string; and MORE, which displays data one screen at a time. The witty technical writers at Microsoft go on to say, "You can easily add your own filter to the filters that have been supplied; just write a program that reads its input from the standard input device, and writes its output to the standard output device." You can easily build your own jetliner, too. Just hook some wings and a fuselage onto some engines, and there you are!

DDJ to the rescue once again. This month we present a detailed example of how to write a filter, with a little program called CLEAN. I previously published this program in a different form in *Softalk/ IBM*, and it was modified into a filter by Bob Taylor of Buffalo, New York. CLEAN can transform almost any kind of text input stream into a stripped-down output file. It may be used, for example, to massage a WordStar docume7< file or other word processing file into a form that EDLIN can cope with. It does this by stripping the high bit off of all characters, expanding tabs to spaces, and deleting all control codes except for form feeds, line feeds, and carriage returns.

In the command line for CLEAN, you must specify the source and destination for the text; otherwise it will simply read the default standard input (the keyboard) and write to the default standard output (the video display). For example, to filter the WordStar document file "MYFILE.WS" and leave the result in "MYFILE.TXT," you would enter:

```
A>CLEAN <MYFILE.WS
   >MYFILE.TXT
```

Note that the original file, MYFILE .WS, is unchanged. An invaluable application of the CLEAN filter is to rescue assembly language source files. I've found that when you accidentally perform extensive editing on such a source file in WordStar "document" mode instead of "nondocument" mode, the resulting file makes the assembler gag and spit out nasty messages. CLEAN lets you turn the source file back into something the assembler can swallow without losing all those painful hours of editing.

Another handy application for CLEAN is to list a WordStar document file in "raw" form on the printer, complete with print commands, etc.:

```
CLEAN <MYFILE.WS >PRN:
```

CLEAN is a simple program and highly modular, as you will see from the accompanying source code (Listing One, page 87). It illustrates reading from the standard input, writing to the standard output, and sending error messages to the (unredirectable) standard error device. It is also remarkably slow. This Macintosh-like performance can be ascribed to the fact that the input and output streams are being treated as character devices. Even when files are being processed, two calls are made to MSDOS for every character filtered. You will find that if you change the "get_char" and "put_char" routines to perform 1024-byte reads and writes, and block/deblock the data internally to the CLEAN program, performance becomes extremely spiffy.

Next month, we'll publish the much more sophisticated "TK" filter in this column. TK, contributed by Jim Mott, is a powerful token parser with many runtime options.

Sizing RAM under MSDOS

In a previous column (*DDJ* No. 90, April 1984), I published a rather circuitous method of finding the amount of available RAM under MSDOS 2.0. Several readers wrote in to take me to task for this item and showed me that there is a much more direct way to get the same result. Billy Smith's letter says it all:

"On the subject of sizing RAM under MSDOS, I know of a very simple solution that I think qualifies as machine independent. In appendix E of the DOS manual we have an explanation of how DOS prepares programs for execution. The preparation, besides loading the file itself into RAM, consists of:

(1) DOS setting up a PSP (Program Segment Prefix) just below the loaded program, and
(2) DOS initializing some of the registers.

"A careful examination of the structure of the PSP shows that the word at

PSP offset 2 contains the paragraph address of the first paragraph following the end of contiguous RAM. This is incredibly unclear in the manual, and it was only by test that I became certain of this fact.

"The manual fails to explicitly give the address of this value in the PSP. It must be deduced from an ambiguous diagram (page E-8). The value, which is called Top of Memory (not too bad a name), is footnoted as follows:

> First segment of available memory is in segment (paragraph) form (for example, hex 1000 would represent 64K).

"It seems that what they mean by 'available memory' is in fact 'unavailable memory,' i.e., the paragraph address of the first paragraph of memory past the end of contiguous RAM. At any rate, this fact of the DOS environment is readily available to every COM and EXE file at load time. I hope this makes sizing of memory a little easier for you and other readers of *DDJ* !"

Many thanks to Billy and the several other readers who provided this helpful information.

More Microsoft Assembler Warnings

While writing this month's column, I stumbled on a new discrepancy between the manual and the Macro Assembler. The manual leads you to believe that the following code is correct:

```
LES AX,DOUBLEWORD PTR
    [BP + 12]
```

In practice, that code causes the assembler to cough up a funny error message. What you need to write is:

```
LES AX,DWORD PTR [BP + 12]
```

The reserved token DWORD isn't mentioned anywhere in the manual as far as I can tell.

Bill Payne of Sandia Labs was kind enough to send a list of further "features" of the Microsoft Macro Assembler. These items were apparently collected on a CompuServe bulletin board by John Chapman and have passed through the hands of an unknown number of intermediate benefactors. I have verified them all again before this appearance in *DDJ*.

(1) The 8088 does not support a comparison of immediate data with a segment register. If you write
CMP ES,0
the assembler will produce the opcode for
CMP AX,0
and does not generate any error message.

(2) Most instructions with missing operands generate error messages; however, the instruction
MOV AX,
produces the machine code for
MOV AX,0
and no error message. Apparently, the same thing will happen for all instructions that can use a general register as the destination and immediate data as the source. Zero is always used for the missing operand.

(3) Erroneous code will be generated if square brackets denoting indexed addressing mode are omitted in certain operations, even though an error message would be expected. If you write
MOV BYTE PTR ES:DI,'$'
the assembler will generate the machine code for
MOV BYTE PTR ES: [7],'$'
rather than what was intended, which was
MOV BYTE PTR ES: [DI], '$'
This appears to occur because the assembler finds DI in its symbol table, equated to the register triplet 111B, and substitutes the 7 as though the programmer had written
DI EQU 7

(4) Even if a RADIX pseudo-op has been used to specify the default base for data values, the assembler will still check the last character for a valid radix specification and use it if present. For example,
.RADIX 16
SUB BX,0B
generates the machine code
83 EB 00
which is incorrect for X'0B', while
SUB BX,11d or SUB BX,0Bh
are both correct and will generate the machine code
83 EB 0B
which is the intended instruction.

(5) Data entry errors. Always scan comments in a newly entered program to make sure that each is preceded by a semicolon rather than a colon. The assembler will assume that the colon denotes a label and will generate the spurious error message "Open Procedures." Review labels in failing programs: the branch table sequence below provides both correct and incorrect examples.

```
            Correct code . : .
            JMP CS:JUMLIST[BX]
            . . .
            . . .
JUMLIST     DW routine1
            DW routine2
            Incorrect code . : .
            JMP CS:JUMLIST [BX]
JUMLIST:    DW routine1
            DW routine2
```

(6) Pseudo-operations. The XLIST pseudo-op will be ineffective during both passes if the command line parameter /D is specified for the assembly. Also, the assembler will not correctly resolve "identity" type definition errors in the EQUATE pseudo-op.
LOOPIT EQU LOOPIT
will cause the assembler to hang.

Hex to ASCII Conversion

Jim Garinger of Hermosa Beach, California, writes: "In the June *DDJ* 16-Bit Toolbox, you included a subroutine to convert a binary word to its hex ASCII equivalent. It seems to be typical of the methods used by my friends with 8-bit systems. Trying to do this same thing myself with the 8086 assembler instructions, I hit upon another way to do it and would like to share it with you. The listing is self-explanatory and lists the number of bytes of code and the clocks (you might want to check me on them) from the Intel manual, *IAPX 88 Book*. This, by the way, has proven to be my invaluable reference when doing 8086 assembler work.

"I would like to point out that further gains in speed can be made with this routine, using my original coding meth-

ods, by coding the call to hexbyte in line twice instead of as a near call. For the five bytes of added size penalty, there is a fair percentage of gain in clock speed. And nobody seems concerned about size anymore anyway.

"I did assemble both of our routines to check actual size, but I am afraid I can't guarantee the accuracy of the clocks or the addition (I ran out of fingers and toes early on)."

Jim's subroutine accompanies this column as Listing Two (page 90).

DDJ

16-Bit Toolbox (Text begins on page 84)

Listing One

```
1                                       name    clean
2                                       page    55,132
3                                       title   'CLEAN - filter text file'
4                               ;
5                               ; CLEAN - a DOS 2.0 filter for word processing document files.
6                               ;
7                               ; Originally written by Ray Duncan
8                               ; Converted to DOS 2.0 filter by Bob Taylor.
9                               ;
10                              ; This program reads text from the standard input device and writes
11                              ; filtered and transformed text to the standard output device.
12                              ;
13                              ;       1.  High bit of all characters is stripped off.
14                              ;       2.  Tabs are expanded.
15                              ;       3.  Removes all control codes except for line
16                              ;           feeds, carriage returns, and form feeds.
17                              ;       4.  Appends an end-of-file mark to the text, if
18                              ;           none was present in the input stream.
19                              ;
20                              ; Can be used to make a WordStar file acceptable for
21                              ; other screen or line editors, and vice versa.
22                              ;
23                              ;
24      = 000D                  cr      equ     0dh             ; ASCII carriage return
25      = 000A                  lf      equ     0ah             ; ASCII line feed
26      = 000C                  ff      equ     0ch             ; ASCII form feed
27      = 001A                  eof     equ     01ah            ; End-of-file marker
28      = 0009                  tab     equ     09h             ; ASCII tab code
29
30      = 0080                  command equ     80h             ; buffer for command tail
31
32                              ; DOS 2.0 Pre-Defined Handles
33
34      = 0000                  stdin   equ     0000            ; standard input file
```

(Continued on next page)

Listing One

```
35      = 0001                  stdout  equ     0001            ; standard output file
36      = 0002                  stderr  equ     0002            ; standard error file
37      = 0003                  stdaux  equ     0003            ; standard auxilliary file
38      = 0004                  stdprn  equ     0004            ; standard printer file
39
40      0000                    cseg    segment para public 'CODE'
41
42                                      assume  cs:cseg,ds:cseg
43
44      0100                            org     100H            ; start .COM at 100H
45
46      0100                    clean   proc    far             ; entry point from PC-DOS.
47      0100  1E                       push    ds              ; push a long return back
48      0101  33 C0                    xor     ax,ax           ; to DOS onto the stack.
49      0103  50                       push    ax
50
51      0104  E8 0160 R         clean3: call    get_char        ; get a character from input.
52      0107  24 7F                    and     al,7fh          ; turn off the high bit.
53      0109  3C 20                    cmp     al,20h          ; is it a control char?
54      010B  73 1D                    jae     clean4          ; no.  write it to output.
55      010D  3C 1A                    cmp     al,eof          ; is it end of file?
56      010F  74 4B                    je      clean6          ; yes, go write EOF mark and exit.
57      0111  3C 09                    cmp     al,tab          ; is it a tab?
58      0113  74 2D                    je      clean5          ; yes, go expand it to spaces.
59      0115  3C 0D                    cmp     al,cr           ; is it a carriage return?
60      0117  74 08                    je      clean35         ; yes, go process it.
61      0119  3C 0A                    cmp     al,lf           ; is it a line feed?
62      011B  74 04                    je      clean35         ; yes, go process it.
63      011D  3C 0C                    cmp     al,ff           ; is it a form feed?
64      011F  75 E3                    jne     clean3          ; no. discard it.
65      0121                    clean35:
66      0121  C7 06 0193 R 0000        mov     column,0        ; if it's a legit ctrl char,
67      0127  EB 05 90                 jmp     clean45         ;  we should be back at column 0.
68
69      012A  FF 06 0193 R      clean4: inc     column          ; if it's a non-ctrl char,
70      012E                    clean45:                        ;  col = col + 1.
71      012E  E8 0178 R                call    put_char        ; write the char to output.
72      0131  73 D1                    jnc     clean3          ; if OK, go back for another char.
73
74      0133  BB 0002                  mov     bx,stderr       ; not OK.  Set up to show error.
75      0136  BA 0195 R                mov     dx,offset err_msg
76      0139  B9 0018 90               mov     cx,err_msg_len  ; error = Disk full.
77      013D  B4 40                    mov     ah,40h          ; write the error message
78      013F  CD 21                    int     21h             ; to the standard error device. (CON:)
79      0141  CB                       ret                     ; back to DOS.
80
81      0142  A1 0193 R      clean5: mov     ax,column       ; tab code detected, must expand
82      0145  99                       cwd                     ; expand tabs to spaces.
83      0146  B9 0008                  mov     cx,8            ; divide the current column counter
84      0149  F7 F9                    idiv    cx              ; by eight...
85      014B  2B CA                    sub     cx,dx           ; eight minus the remainder is the
86      014D  01 0E 0193 R             add     column,cx       ; number of spaces to send out to
87      0151                    clean55:                        ; move to the next tab position.
88      0151  51                       push    cx
89      0152  B0 20                    mov     al,20h
```

```
0154  E8 0178 R                   call    put_char        ; send an ASCII blank
0157  59                         pop     cx
0158  E2 F7                      loop    clean55
015A  EB A8                      jmp     clean3

015C  E8 0178 R             clean6: call    put_char        ; write out the EOF mark,
015F  CB                            ret                     ; and return to DOS.

0160                        clean   endp


0160                        get_char proc near
0160  BB 0000                      mov     bx,stdin                ; get chars from std. input
0163  B9 0001                      mov     cx,1                    ; # of chars to get = 1
0166  BA 0191 R                    mov     dx,offset input_buffer  ; location = input_buffer
0169  B4 3F                        mov     ah,3fh
016B  CD 21                        int     21h                     ; do the function call
016D  0B C0                        or      ax,ax                   ; test # of chars returned
016F  74 04                        jz      get_char1               ; if none, return EOF
0171  A0 0191 R                    mov     al,input_buffer         ; else, return the char in AL
0174  C3                           ret
0175                        get_char1:
0175  B0 1A                      mov     al,eof                  ; no chars read, return
0177  C3                           ret                             ; an End-of-File (EOF) mark.
0178                        get_char endp

0178                        put_char proc near
0178  A2 0192 R                    mov     output_buffer,al        ; put char to write in buffer.
017B  BB 0001                      mov     bx,stdout               ; write to std. output
017E  B9 0001                      mov     cx,1                    ; # of chars = 1
0181  BA 0192 R                    mov     dx,offset output_buffer ; location = output_buffer
0184  B4 40                        mov     ah,40h
0186  CD 21                        int     21h                     ; do the function call
0188  3D 0001                      cmp     ax,1                    ; check to see it was really done.
018B  75 02                        jne     put_char1
018D  F8                           clc                             ; really done. return carry = 0
018E  C3                           ret                             ; as success signal.
018F                        put_char1:
018F  F9                           stc                             ; not really done. return carry = 1
0190  C3                           ret                             ; as error signal (device is full).
0191                        put_char endp

0191  00                    input_buffer    db      0
0192  00                    output_buffer   db      0

0193  0000                  column          dw      0

0195  0D 0A                 err_msg         db      cr,lf
0197  63 6C 65 61 6E 3A                     db      'clean: Disk is full.'
      20 44 69 73 6B 20
      69 73 20 66 75 6C
      6C 2E
01AB  0D 0A                               db      cr,lf
= 0018                      err_msg_len     equ     (this byte)-(offset err_msg)

01AD                        cseg    ends
```

(Continued on next page)

Listing One

```
146
147                                 end     clean
```

End Listing One

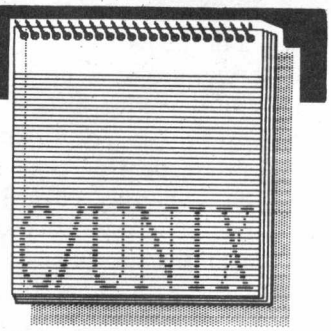
Listing Two

Jim Garinger's binary to hex ASCII subroutine.

```
; Routine to convert 16-bit binary
; word to hexadecimal ASCII string.

; Call with:     AX = 16-bit binary value
;                SI = first-byte-address for
;                     resulting ASCII string
;
; Returns:       nothing
;
; Destroys:      AX, BX, CX, SI, clears direction flag
;
; In each comment, the first number is the number of
; bytes of machine code, and the second is the number
; of machine cycles to execute that instruction.

hexconv     proc        near
                                ;45 bytes/276 clocks
            cld                 ; 1   2
            mov         cl,4    ; 2   4 shift count
            mov         bx,offset hextbl
                                ; 3  12 point to table for translations
            mov         ch,al   ; 2   2 store for converting 2nd half

                                ; convert upper half of
                                ; word to hex ASCII

            call        hexbyte ; 3  19  (98 clocks)
            mov         ah,ch   ; 2   2 get the word

                                ; convert lower half of
                                ; word to hex ASCII

            call        hexbyte ; 3  19  (98 clocks)
            ret                 ; 1  20
                                ;    80 (+ 2*98 = 276 total clocks)
hexbyte     proc        near
            xor         al,al   ; 2   3 clear for positioning
            shr         ax,cl   ; 2  20 low nibble into al
            shr         al,cl   ; 2  20 position lower nibble
            xlat        hextbl  ; 1  11 convert lower nibble
            xchg        al,ah   ; 2   2 get higher nibble
            xlat        hextbl  ; 1  11 convert higher nibble
            stosw               ; 1  11
            ret                 ; 1  20

                                ;conversion table to hex ASCII
hextbl      db          "0123456789ABCDEF"

hexbyte     endp
hexconv     endp
```

End Listings

C/UNIX PROGRAMMER'S NOTEBOOK



By Anthony Skjellum

The thrust of this column is to provide examples of scientific uses for the C programming language. My aim is to present routines useful in conjunction with scientific programming; some are general and others implement specific numerical algorithms. I intend this column for several specific audiences. The first audience comprises those diehard programmers who continue to produce useful programs in outdated languages. Not only is this of greater expense to themselves, but it also cheats others by locking them into Fortran or other old-fashioned languages. For them, I want to illustrate the elegance and versatility of C.

The second audience (which may also include the first) is those users interested in scientific applications who may not have used C for this purpose. This column should demonstrate that C is completely acceptable for such purposes, and the code shows how generally concepts can be presented. (I make no attempt to survey scientific applications where C could be used but merely include some nontrivial examples.) Finally, for those readers who don't fit into the above categories, the general purpose routines should still prove interesting.

The column presents three programming systems. The first is a set of general purpose subroutines designed to simplify the process of user – program interaction and to provide a straightforward means for handling erroneous input. The input mechanisms are not particularly sophisticated but emphasize structure in the user's program.

All the code presented with this article is copyright © 1983, 1984 by the California Institute of Technology (Caltech), Pasadena, CA 91125. All rights reserved. This code may be freely distributed, used for all non-commercial purposes, but may not be sold.

The routines make range checking so automatic that the programmer has no excuse for omitting such checks, regardless of how quickly a program must be completed. Providing these routines to novice programmers in a classroom environment has eliminated a lot of frustration over using the scanf( ) function.

The second and third programming systems illustrate Runge-Kutta integration. The Runge-Kutta formalism is a standard numerical technique for handling the numerical integration of one or more first-order ordinary differential equations. Interested readers may wish to consult the book *Numerical Analysis* by Richard L. Burden, et al. (Prindle, Weber, Schmidt Publishers, 2nd ed., 1981). This is the source for the algorithms presented in the code and is also a fine reference for introductory numerical methods. I chose the Runge-Kutta routines as examples because of their widespread use in scientific work.

Acknowledgements

The general purpose library has received extensive use by Caltech students during the past school year. Thanks are due to Scott Lewicki who discovered a couple of minor details that caused major errors.

The Runge-Kutta code was developed by Michael J. Roberts and my-

```
int iinp(prompt,cflag,low,high);
char *prompt: optional prompt string to be printed before input
char cflag:   checking flag: if non-zero, range checking is performed
int low
int high:     low, high are (inclusive) range checking values
iinp( ) repeats input until a valid number is entered; the valid number is the function's return value.

double finp(prompt, cflag, low, high);
char *prompt: optional prompt string to be printed before input
char cflag:   checking flag: if non-zero, range checking is performed
double low
double high:  low, high are (inclusive) range checking values
finp( ) repeats input until a valid number is entered; the valid number is the function's return value

len = sinp(prompt, string, length);
int len:      length of entered string
char *prompt: optional prompt string to be printed before input
int length:   maximum length of input string
sinp( ) ignores leading spaces.

retn = cinq(prompt);
int retn:     1 —> 'Y' was typed, 0 —> 'N' was typed
char *prompt: optional prompt string to be printed before input

retn = display(fname);
int retn:     0 —> success, -1 —> failure
char *fname:  null-terminated name of file
display( ) prints the specified file on the standard output device.
```

Table I
GPR Calling Sequences

self. Mr.Roberts developed the major portion of the RKSYS (multiple equations) routines as a project for Caltech's Introduction to Computational Physics course. Approximately 60 hours were spent developing, testing, and debugging this code. Mr. Roberts also wrote the original version of the documentation for RKSYS, included in modified form as Table III (page 94).

GPR: General Purpose Routines

The general purpose library consists of five subroutines. Four of these subroutines deal with input. The fifth is a simple facility for printing files to the console. This latter routine, called display( ), will be considered separately from the input functions.

The other routines are iinp( ), finp( ), sinp( ), and cinq( ). The first three provide integer, floating-point, and string input, respectively. The last is a yes-no question processor. Table I (page 92) provides the exact calling sequences for each of the routines.

Traditionally user input consists of a sequence of lines, such as the sequence for inputting an integer shown in Figure 1 (at right).

Entering this sequence repeatedly can be rather tedious, so error checking is often omitted. This practice leads to programs that do not handle user mistakes intelligently. The GPR routines allow you to replace the above sequence with a single line of code:

```
input = iinp ("Enter input variable
    —>",1,MIN,MAX);
```

Since using the GPR input functions is easier than using scanf( ), this variety of function should be well received and may be used in lieu of scanf( ) for most purposes. A further advantage of these functions is that they unclutter the user's program. More sophisticated checking must still be included explicitly; the input functions only do range checking.

The display( ) function was included to encourage users to provide on-line help/documentation along with their programs. This function allows users to print out additional text whenever appropriate, creating a trivial on-line help facility; a help feature is almost always appropriate but usually is omitted.

```
#define MIN 10
#define MAX 100
. . .
int input;
. . .
while (1)       /* input loop */
{
      printf("Enter input variable —> ");
      if (scanf("%u",&input) != 1)      /* get variable */
      {
            drain( );      /* drain spurious characters */
            continue;      /* skip range checks */
      }
      /* do range checking */
      if ((input >= MIN) && (input <= MAX))
            break;         /* we are done */
      printf("\nNumber out of range\n");
} /* keep looping until scanf( ) can read a variable */
```

Figure 1

File: RK4.C Subroutine library: Listing Three.

The C language call is as follows:

```
rk4(function,a,b,n,alpha,t,w);
```

where:

function	returns the righthand side f(y,t) of the system
a	is the start of the interval of integration
b	is the end of the interval of integration
n	is the number of integration steps
alpha	is the initial value y(0) = alpha
t	is the array where the times will be stored
w	is where the approximations to y will be stored

The formal C definitions for the functions and its parameters are:

1. rk4( ): n step integrator

```
rk4(function,a,b,n,alpha,t,w)
double (*function)( );    /* function giving f(t,y) */
double a;                 /* beginning of interval */
double b;                 /* end of interval */
int n;                    /* number of steps in interval */
double alpha;             /* initial condition for y */
double t[ ];              /* array for returning T[i] values */
double w[ ];              /* array for returning W[i] values */
```

2. rk4_1( ): 1 step integrator

```
rk4_1(function,h,time,yapprox)
double (*function)( );    /* pointer to function to integrate */
double h;                 /* step size */
double time;              /* current time step */
double *yapprox;          /* current approximation of function */
```

Table II
RK4: Runge-Kutta Integrator

Listing One (page 96) shows the GPR routines. The current list is not exhaustive but intended only to suggest a trend for additional routines.

RK4: Runge-Kutta Algorithm

Now we will consider a scientific application of C: single equation Runge-Kutta integration. Before introduction of the code, some background is required.

We start with a differential equation in the canonical form

$$y' = f(y,t)$$

where y′ is the first derivative of y with respect to t and f(y,t) is a piece-wise continuous function of its arguments. In addition to the differential equation, we are given an initial condition

$$y(t=0) = y0$$

where y0 is some real constant (e.g., 35.1). These two equations uniquely specify the solution y(t), which is as yet unknown. The need for numerical techniques arises when the differential equation cannot be solved analytically.

Many possible numerical approaches can solve this equation, but considering all of them is beyond the scope of the current discussion. It is sufficient to state that a technique called RK4 (Runge-Kutta fourth order) will solve the equation numerically with known error characteristics. Listing Two (page 98) presents the solution of a typical equation

$$y'(t) = 1 + y$$

with the initial condition

$$y(t = 0) = 5.0$$

Since this equation can also be solved analytically, a comparison with the exact solution will demonstrate the error characteristics of the method. The analytical solution turns out to be

$$y(t) = t + 5.0*\exp(t)$$

This result can be deduced by inspection.

The Runge-Kutta algorithm and code are presented in Listing Three (page 101). Readers interested in more information should consult the Burden book. Calling sequences are described in Table II (page 93).

RKSYS: Systems of Differential Equations

Imagine now that we have a set (system) of N first-order ordinary differential equations:

$$y_i'(t) = f(t, y_i, y_1, y_2 \ldots y_N)$$
$$(i = 1,2,\ldots N)$$

This problem is useful for solving other systems too, since sets of linear differential equations involving higher order derivatives can be transformed to larger systems of the above variety (see Burden). Thus, a program that can solve the above system has reasonably wide applicability.

Unfortunately, the problem is more complicated for N equations than for one equation, as is evident from Table III (page 95). This table describes the more general Runge-Kutta software, and Listing Four (page 106) contains the actual code. Listings Five and Six (pages 110–114) are example programs that use rk4n( ) to solve small systems of equations. Listing Five implements the same problem as Listing Two but uses rk4n( ) instead of rk4( ) to perform the integration.

Using the Integrator

The rk4n( ) subroutine will solve a sys-

Files: RKS.C — Subroutine library: Listing Four
RKST1.C — Test program 1: Listing Five
RKST2.C — Test program 2: Listing Six

The format of the C language call is as follows:

```
rk4n(function,wsource,wstore,m,a,b,n,alpha,t,kuttas);
```

where:

function is a pointer to the function that will return the first derivatives of each function in your system. It will be called as:

```
function(j,i,tval,rk_comp)
```

where:

- **j** is the current time step
- **i** is the current function number (0, 1, . . . , n – 1).
- **tval** is the current time value
- **rk_comp** is a pointer to a function that your derivative function must call as:

```
rk_comp(n,j,i)
```

where:

n is the function number in the system (0, . . . n – 1) of the function you wish to evaluate
j is the time step
i is the current function number

wsource is a pointer to the function that will return a W value (which will have been stored by your WSTORE routine—one need worry only about storing and returning these values, not the values themselves). It will be called as:

```
wsource(j,i)
```

where:

- **j** is the current time step
- **i** is the current function number

wstore is a pointer to the function that will store values of W (see wsource above). It is called as:

```
wstore(j,i,value)
```

where:

- **j** is the current time step

Table III

(Continued on next page)

tem of first-order differential equations as specified by the calling program, using the Runge-Kutta fourth-order integrator (described in Burden, pages 239 – 240; refer also to page 205).

The calling program must provide information for the rk4n( ) subroutine before it can solve the system of differential equations. This information consists of:

- The first derivatives of each of the functions in the system
- Subroutines to store and retrieve values as the functions are integrated
- Initial conditions for each of the functions
- The range over which the functions will be integrated
- Storage areas for the time and intermediate "k" value arrays

The information must be provided in a specific order and format, as described in Table III.

By studying the Runge-Kutta routines, the reader will notice the central importance of pointers to functions in the organization of the code. Using this concept effectively allows the software to be completely divorced from the specifics of the user's program.

Conclusions

This article presented three sets of example routines to demonstrate how scientific applications and related software are actually implemented in C. Studying the examples should help the reader to gain insight into the use of C for similar undertakings. DDJ

(Listings begin on page 96)

	i is the current function number value is the value to be stored for that location Note:wsource(j,i) should be equal to "value" after wstore(j,i,value).
m	is the number of equations in your system
a	is the starting time value for the interval
b	is the ending time value for the interval
n	is the number of points into which the interval is to be broken
alpha	is a one-dimensional array of the initial values. The value alpha[0] is the first function at time = a, alpha[1] is the second, etc.
t	is a one-dimensional array where rk4n( ) will store the time values as it calculates them. It must be at least as large as n.
kuttas	is a two-dimensional array, the size of whose second element is 4 (i.e., Kuttas[ ][4]). It will be the storage area for "k" values as they are calculated.

The formal C definitions for the function and its parameters are:

```
double rk4n(function,wsource,wstore,m,a,b,n,alpha,t,kuttas)
double (*function)( );
double (*wsource)( );
double (*wstore)( );
int m;
double a;
double b;
int n;
double alpha[ ];
double t[ ];
double kuttas[ ][4];
```

Comments on array sizes, (*wsource)( ), (*wstore)( ):
The (*wstore)( ) function should trap out-of-bound storage requests that result as a natural part of the rk4n( ) algorithm. A more elegant solution is to make the array size used by ((one greater than the number of steps.

Table III
RK System Solver (RKSYS)

Listing One

```
/*
        gpr.c                                   created: 01-Oct-83

        general purpose utility routines for use with C to simplify
        user/program interaction.

        by Anthony Skjellum

        Copyright 1983, 1984 (c) Caltech.  All rights reserved.
        This subroutine library may be distributed freely and
        used for all non-commercial purposes but may not be sold.

        Routines:

                iinp():  integer input w/ prompt, range check + retry
                finp():  float   input w/ prompt, range check + retry
                sinp():  string  input w/ prompt
                cinq():  yes/no question processor w/ prompt + retry
                display(): display a file

        updated: 20-Jul-84

        testing information: this code was tested with Aztec C 1.05j
*/

#include <stdio.h>

/* Unix flag (used by display()) */
/*
#define UNIX 1
*/
#ifndef UNIX
#define TEOF    26      /* ^Z */
#endif

/* general purpose subroutines: */

/* iinp(): integer input with range checking, prompt and retry */

iinp(prompt,cflag,low,high)
char *prompt;
char cflag;
int low;
int high;
{
        int ival;

        while(1)
        {
                printf("%s",prompt);
                if(scanf("%d",&ival) < 1)
                        while(getchar() != '\n')
                                ;

                if((!cflag)||(ival >= low)&&(ival <= high))

                        break;
                                /* no checking, or within bounds */

                printf("\nValue out of range, try again...\n");
        }

        return(ival); /* return the value */

} /* end iinp() */
```

(Continued on next page)

Listing One

```
/* finp(): floating point input with range checking, prompt and retry */

double finp(prompt,cflag,low,high)
char *prompt;
char cflag;
double low;
double high;
{
        double fval;

        while(1)
        {
                printf("%s",prompt);
                while(scanf("%lf",&fval) < 1)
                        while(getchar() != '\n')
                                ;

                if((!cflag)||(fval >= low)&&(fval <= high))
                        break;
                                /* no checking, or within bounds */

                printf("\nValue out of range, try again...\n");
        }

        return(fval); /* return the value */

} /* end finp() */

/* subroutine sinp(): input a string with prompt, length limit */

sinp(prompt,string,length)
char *prompt;
char *string;
int length;
{
        int len;                        /* length of actual string input */
        char chr;
        char *fgets();                  /* string input function */

        printf("%s",prompt);                    /* display the prompt */
        while(isspace(chr = getchar()))
                ;
        ungetc(chr,stdin);
        fgets(string,length,stdin);     /* input the string */

        if((len = strlen(string)))
                string[strlen(string)-1] = '\0';

        return(len);
} /* end sinp() */

/* subroutine cinq(): yes no question processor with prompt, retry  */

cinq(prompt)
char *prompt;
{
        char chr;

        while(1)
        {
                printf("%s",prompt);

                do /* drain spurious 'white space' */
                {
                        chr = tolower(getchar()); /* use first char */
                }
                while(isspace(chr));

                if((chr == 'y')||(chr == 'n')) break;

                printf("\nRespond with Y or N, please try again...\n");
```

```
        }

        return((chr == 'y') ? 1 : 0);
}

/* display(): subroutine to print an ascii file on the console */

display(fname)
char *fname;
{
        char c;         /* character to output */
        FILE *disp;

        if((disp = fopen(fname,"r")) == NULL)
                return(-1);     /* can't open file */

        while((c = getc(        /* And the last one */

        *yapprox += (k1 + 2.0*(k2 + k3) + k4)/6.0; /* new approx */
}
```

End Listing One

Listing Two

```
/*
        program:        rktest1.c
        created:        03-Nov-83
        by:             A. Skjellum

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        updated:        16-Nov-83
        purpose:        illustrate the use of rk4 program

        uses:           rk4.c

        summary:
                integrates the differential equation:

                y'(t) + y(t) = t + 1
                y(0) = 5.0.

                for which the exact solution:

                y(t) = t + 5exp(-t) is known.

*/

/* constants */

#define YZERO   5.0     /* initial value for y */
#define TSTART  0.0     /* starting time for integration */
#define TEND    10.0    /* ending time for integration */
#define STEPS   80      /* 40 steps in integration */


/* subroutines: */

/* exact(): returns exact solution value, given t */

double exact(t)
double t;
{
        extern double exp();    /* exponential function */
```

(Continued on next page)

Listing Two

```
        if(t)
                return((t + YZERO*exp(-t)));

        return(YZERO);
}

/* fn(t,y): return f(t,y) given t,y values */

double fn(t,y)
double t;
double y;


{
        /*
                differential equation is y' + y = t + 1

                therefore, f = t + 1 - y.

        */

        return(t + 1.0 - y);
}

/* solutn(): print solution step at console */

solutn(t,y)
double *t;      /* pointer to t value */
double *y;      /* pointer to y value */
{
        printf("t = %7.3e, y = %7.3e, y_exact = %7.3e, diff = %7.3e\n",
                *t,*y,exact(*t),*y - exact(*y));
}

/* main program: */

main()
{
        /* external declarations */

        double fn();    /* ensure that this is typed as double */

        /* local variables: */

        register int i;

        double yarray[STEPS],tarray[STEPS];
                /* integrated solution stored here */


        /* begin code: */

        printf("\n\nrktest1.c   as of 03-Nov-83\n\n");
        printf("Integrates: y' + y = 1 + t     for\n\n");
        printf("t = %7.3e to %7.3e, with %u steps\n\n",
                TSTART,TEND,STEPS);

        /*
                integrate the answer from t = 0 to t = 10 sec
                80 points.
        */

        rk4(fn,TSTART,TEND,STEPS,YZERO,tarray,yarray);
                        /* compute the answers */
        for(i=0;i<STEPS;i++)    /* print solution */
                solutn(tarray+i,yarray+i);

       printf("\n\nEnd of execution\n\n");
}
```

End Listing Two

Listing Three

```
/*
        Runge - Kutta order 4 Algorithm

        Creation date:  31-Oct-83
        Author:         Mike Roberts

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        This algorithm is described in detail on page 205 of
        Burden, Richard L.:  Numerical Analysis.

        To approximate the solution of the initial value problem
                y'=f(t,y),  a<=t<=b, y(a)=alpha,
        at (N+1) equally spaced numbers in the interval [a,b]:
        INPUT endpoints a,b; integerg N; initial condition alpha.
        OUTPUT approximation w to y at the (N+1) values of t.

Step 1:
        Set     h=(b-a)/N;
                t=a;
                w=alpha;
        Output (t,w).
Step 2:
        For i=1,2,...,N do Steps 3-5:

        Step 3:
                Set     K1=hf(t,w);
                        K2=hf(t+h/2,w+K1/2);
                        K3=hf(t+h/2,w+k2/2);
                        K4=hf(t+h,w+K3).
        Step 4:
                Set w=w+(K1+2K2+2K3+K4)/6;      (Compute w[i].)
                        t=a+ih.                 (Compute t[i].)
        Step 5:
                Output (t,w).

Step 6:
        Stop.


*/


#define FALSE 0
#define TRUE 1

rk4(function,a,b,n,alpha,t,w)

double (*function)();   /* function giving f(t,y) */
double a;               /* beginning of interval */
double b;               /* end of interval */
int n;                  /* number of steps in interval */
double alpha;           /* initial condition for y */

double t[];             /* array for returning T[i] values */
double w[];             /* array for returning W[i] values */

{
        register int i; /* counter for integration steps */
        double h;       /* stepsize */
        double time;
        double yapprox; /* approximation for y value  */
```

(Continued on next page)

Listing Three

```
        /*  STEP 1:  Initialization  */

        h = (b-a) / (double)n ;          /* Compute stepsize */
        time = a;                        /* Initialize time */
        yapprox = alpha;                 /* Start with the approximation
                                            equal to the initial value */

        for (i=0; i<n; i++)              /* Main integration loop */
        {
                if(i)   /* if not first time, call the integrator */
                        rk4_1(function,h,time,&yapprox);

                        /* Pass the function pointer, the h, time, and
                                yapprox values, and the pointers to
                                the current positions in the T and W
                                matrices */

                time = a + h*(double)i;  /* compute time */
                t[i] = time;             /* also save it */
                w[i] = yapprox;          /* store value for function */
        }
}

/*  This is the RK4 integrator portion.  It performs one step of the
        integration, and is called on each step from the RK4 loop.

        function = pointer to function to integrate
        h = stepsize
        time = current time location
        yapprox = current w (function approximation)

*/

rk4_1(function,h,time,yapprox)

double (*function)();            /* Pointer to function to integrate */
double h;                        /* Step size */
double time;                     /* Current time step */
double *yapprox;                 /* Current approximation of function */

{
        double k1, k2, k3, k4;  /* Temporary values in RK calculation */

        k1 = h * (*function)(time,*yapprox);     /* Evaluate first approx */
        k2 = h * (*function)(time+h/2.0, *yapprox+k1/2.0);
                                                 /* Evaluate second approx *


        k3 = h * (*function)(time+h/2.0, *yapprox+k2/2.0);
                                                 /* And the third */
        k4 = h * (*function)(time+h, *yapprox+k3);
                                                 /* And the last one */

        *yapprox += (k1 + 2.0*(k2 + k3) + k4)/6.0; /* new approx */
}
```

End Listing Three

Listing Four

```
/*
        rk4n.c                          created: 07-Nov-83
        authors:                        A. Skjellum,
                                        M. Roberts

        updated:                        14-Nov-83
                                        by MJR

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        Purpose:
        integrate M first order differential equations

        y'[i] = f[i](t;y[j=1...M])

        M equations.

        algorithm:      see Burden, Faires, Reynolds, p. 239-240
                        also see p. 205

                1. interval t = [a,b]
                2. choose N > 0 as as partition of interval (N steps)
                3. define step size h = (b-a)/N.
                4. Initial conditions: (denote w[i,j] as approxes to y's)

                        w[i,0] = alpha[i]
                        means: ith w at time zero is set to initial value
                               alpha[i].

                5. Computing the w[i,j+1] from w[i,j] is done as follows:

                        loop over i = 1 to M

                            compute k1[i] = h*f[i](t,w[1,j],...,w[M,j])

                        end of loop

                        loop over i = 1 to M

                            compute k2[i] = h*f[i](t+h/2,w[1,j]+.5*k1[1],...,
                                            w[M,j] + .5*k1[M])

                        end of loop

                        loop over i = 1 to M

                            compute k3[i] =h*f[i](t+h/2,w[1,j]+.5*k2[1],...,
                                           w[M,j] + .5*k2[M])

                        end of loop

                        loop over i = 1 to M


                            compute k4[i] =h*f[i](t+h,w[1,j]+k3[1],...,
                                           w[M,j] + k3[M])

                        end of loop

                        loop over i = 1 to M

                            w[i,j+1] = w[i,j] +
                                       {k1[i] + 2*k2[i] + 2*k3[i] + k4[i]}/6

                        end of loop
```

(Continued on next page)

Listing Four

```
*/

double  (*rk_function)();
double  (*rk_source)();
double  (*rk_store)();
double  (*rk_kuttas)[4];
double  (*rk_comp[4])();                /* tells us how to form k's */

/* functions called indirectly by k_calc()  */

double rk_1(n,j,i)      /* to provide compatibility with calling */
int n;
int j;
int i;
{
        return ((*rk_source)(j,n));
}

double rk_23(n,k,j,i)  /*  N is in { 0...M } = argument number.
                           Since we have one argument to FN per equation,
                           N will indicate which we are currently being
                           asked to provide.  Same goes for other RK_x
                           functions below.  */
int n;
int k;
int j;
int i;
{
        return((*rk_source)(j,n) + .5*rk_kuttas[n][k]);
}

double rk_2(n,j,i)
int n;
int j;
int i;
{
        return(rk_23(n,0,j,i));
}

double rk_3(n,j,i)
int n;
int j;
int i;

{
        return(rk_23(n,1,j,i));
}

double rk_4(n,j,i)
int n;
int j;
int i;
{
        return((*rk_source)(j,n) + rk_kuttas[n][2]);
}


/*  Here's the integrator!!! */


double rk4n(function,wsource,wstore,m,a,b,n,alpha,t,kuttas)

double (*function)();    /* pointer to function which returns deriv info */
double (*wsource)();     /* source of w[i,j] values */
double (*wstore)();      /* function which stores w[i,j] values for us */
int m;                   /* number of equations */
double a;                /* start of interval */
double b;                /* end   of interval */
```

```
        int n;                  /* number of points */
        double alpha[];         /* array of initial values */
        double t[];             /* array where we store times */
        double kuttas[][4];     /* n x 4 kuttas (k1,k2,k3,k4 i=1,...m) */
        {
        register int i; /* looping variable */
        register int j; /* looping variables */
        double time;
        double h = (b-a)/(double)n;     /* step size */

        double rk_k2(),rk_k3(),rk_k4(); /* include for emphasis */

        rk_function = function;

        rk_kuttas   = kuttas;
        rk_source   = wsource;
        rk_store    = wstore;

        rk_comp[0]  = rk_1;
        rk_comp[1]  = rk_2;
        rk_comp[2]  = rk_3;
        rk_comp[3]  = rk_4;

        /*  First assign initial values  */

        for (i = 0;i < m;i++)
                (*rk_store)(0,i,alpha[i]);

        /*  Now loop through the necessary loop, calculating
                each K value for each equation in each time step. */



        for(j = 0;j < n;j++)    /* n time steps */
        {
                time = a + h*(double)j;  /* compute time */
                t[j] = time;             /* also save it */

                rk4_1n(m,j,time,h);
        }
}

/* rk4_1n(): compute one solution step for n equations */

rk4_1n(m,j,time,h)
int m;
int j;                          /* time step we're working on */
double time;
double h;
{
        register int k; /* k calculation loop */
        register int i; /* m equations loop */
        double value;   /* temporary */

        for(k=0;k < 4;k++)      /* k compute loop */
                k_calc(m,k,j,time,h);

        for(i=0;i<m;i++)        /* compute new w[i,j]'s j fixed here */
        {
                value = (*rk_source)(j,i) + .166666667*(rk_kuttas[i][0] +
                        2.0*(rk_kuttas[i][1] + rk_kuttas[i][2])
                            + rk_kuttas[i][3]);
                                /* value for w[i,j] */
                (*rk_store)(j+1,i,value); /* save this hard got number */

        }
}

/* k_calc(): compute rk coefficients for fixed j */
```

(Continued on next page)

Listing Four

```
k_calc(m,k,j,time,h)
int m;
int k;
int j;
double time;
double h;
{
        register int i;
        double tval;

        switch(k)
        {
                case 0:
                        tval = time;
                        break;
                case 1:

                case 2:
                        tval = time + .5*h;
                        break;
                case 3:
                        tval = time + h;
                        break;
        }

        for(i=0;i<m;i++)                /*  used to be 1;<=m  */
        {
                rk_kuttas[i][k] = h*(*rk_function)(j,i,tval,rk_comp[k]);
        }
}
```

End Listing Four

Listing Five

```
/*
        program:        rkst1.c
        created:        03-Nov-83
        by:             A. Skjellum

        modified:       14-Nov-83
        by:             M. J. Roberts

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        purpose:        illustrate the use of rk4n program

        uses:           rk4n() (rks.c)

        summary:
                integrates the differential equation:

                y'(t) + y(t) = t + 1
                y(0) = 5.0.

                for which the exact solution:

                y(t) = t + 5exp(-t) is known.

                Integrates the same equation as rktest1
                but using the more general equation solver,
                rk4n().  This run is exactly the same as for
```

```
                rktest1, except that, rather than trying to solve
                exactly the same equation, we will solve a
                "system" of one differential equation.

*/

/* constants */

#define SYSIZE  1       /* number of functions in system */
#define YZERO   5.0     /* initial value for y */
#define TSTART  0.0     /* starting time for integration */
#define TEND    10.0    /* ending time for integration */
#define STEPS   50      /* 50 steps in integration */

/* variables external to all functions */

double wvalue[STEPS+1][SYSIZE];
double yarray[STEPS+1][SYSIZE];
double tarray[STEPS];
        /* integrated solution stored here */


/* subroutines: */

/* exact(): returns exact solution value, given t */


double exact(t)
double t;
{
        extern double exp();    /* exponential function */

        return((t + YZERO*exp(-t)));
}

/* fn(j,i,t,y): return f(t,y) given t,y values */

double fn(j,i,t,y)
int j;
int i;
double t;
double (*y)();
{
        double a,b;             /*  temporary storage space  */

        /*
                differential equation is y' + y = t + 1

                therefore, y' = t + 1 - y.

        */

        a = (*y)(0,j,i);        /*  calculate function
                                Note that the ZERO was passed so as to
                                allow the function to know which argument
                                we are talking about -- in this case,
                                we only need one argument evaluated, so
                                pass it 0 to indicate the first (zeroeth,
                                actually) argument is to be calculated.  */

        b = t + 1.0 - a;        /*  and figure out the rest of it  */
        return(b);

}
```

(Continued on next page)

C/Unix (Listing Continued, text begins on page 92)

Listing Five

```
/* store():  the routine to store away the W values for later reference */

double store(row, col, value)
int row, col;           /*  location to store the value  */
double value;           /*  the actual value to store  */
{
        wvalue[row][col]=value;
        return (value);
}

/* source():  return the  W value referenced by input parameters  */

double source(row,col)
int row, col;           /*  location to look up */


{
        return (wvalue[row][col]);
}

/* solutn(): print solution step at console */


solutn(j,i)
int j,i;        /* element numbers  */
{
        double time;
        double ex;
        double approx;

        time   = tarray[j];
        ex     = exact(time);
        approx = source(j,i);

        printf("t = %7.3e, y = %7.3e, y_exact = %7.3e, diff = %7.3e\n",
                time,approx,ex,approx - ex);
}

/* main program: */

main()
{
        /* external declarations */

        double store(), source();

        double fn();    /* ensure that this is typed as double */

        /* local variables: */

        register int i,j;


        double init[1];         /* initial condition matrix */

        /* begin code: */

        printf("\n\nrktest1.c   as of 03-Nov-83\n\n");
        printf("Integrates: y' + y = 1 + t     for\n\n");
        printf("t = %7.3e to %7.3e, with %u steps\n\n",
                TSTART,TEND,STEPS);

        /*
                integrate the answer from t = 0 to t = 10 sec
                STEPS points.
        */

        init[0] = YZERO;        /* set up initial condition matrix */

        rk4n(fn,source,store,SYSIZE,TSTART,TEND,STEPS,init,tarray,yarray);
                        /* compute the answers for 1 function */
```

```
        /*  Print out solution  */

        for(j=0;j<STEPS;j++)     /* print solution */
                solutn(j,0);
        ,

        printf("\n\nEnd of execution\n\n");
}
```

End Listing Five

Listing Six

```
/*
        program:        rkst2.c
        created:        03-Nov-83
        by:             A. Skjellum

        modified:       14-Nov-83
        by:             M. J. Roberts
        and:            5-Dec-83
        by:             M. J. Roberts

        modified:       25-Jul-84
        by:             A. Skjellum

        Copyright 1983, 1984 (c) California Institute of Technology.
        All rights Reserved.  This program may be freely distributed
        for all non-commercial purposes but may not be sold.

        purpose:        illustrate the use of RKS program
        update:         to test the rk4n() subroutine using a system
                        of two differential equations.

        uses:           rk4n() (rks.c)
        summary:

                integrates the differential equation system:

                u1'(t) = 8u2(t)                 u1(0) = 10
                u2'(t) = 2u1(t)                 u2(0) = 7

                for which the exact solution is known to be:

                u1(t) = 12exp(4t) - 2exp(-4t)
                u2(t) =  6exp(4t) +  exp(-4t)


*/

/* constants */

#define SYSIZE  2       /* number of functions in system */
#define Y1ZERO  10.0    /* initial value for first equation */
#define Y2ZERO  7.0     /* initial value for other equation  */
#define TSTART  0.0     /* starting time for integration */
#define TEND    1.0     /* ending time for integration */
#define STEPS   50      /* 50 steps in integration */

/* variables external to all functions */
```

(Continued on next page)

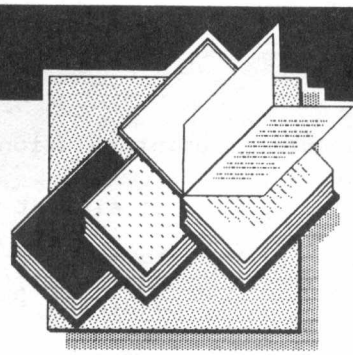
Listing Six

```
double wvalue[STEPS+1][SYSIZE];
double yarray[STEPS+1][SYSIZE],tarray[STEPS];
        /* integrated solution stored here */


/* subroutines: */

/* exact(): returns exact solution value, given t */

double exact(n,t)
int n;          /*  which equation is it?  0 or 1?  */
double t;

{
        extern double exp();    /* exponential function */

        /*  This must find solutions for both U1 and U2.  The
                exact solutions are given in the header comments to
                this program, above.  */

        switch (n)
        {
                case 0:
                        return(12*exp(4*t) - 2*exp(-4*t));
                case 1:
                        return( 6*exp(4*t) +   exp(-4*t));
}       }


/* fn(j,i,t,y):  return f(t,y) given t,y values */

double fn(j,i,t,y)
int j;
int i;
double t;
double (*y)();
{
        switch (i)
        {
                case 0:
                        /*  u1'(t) = 8 * u2(t)  */
                        return(8*(*y)(1,j,i));

                case 1:
                        /*  u2'(t) = 2 * u1(t) */
                        return(2*(*y)(0,j,i));
        }
}


/* store():  the routine to store away the W values for later reference */

double store(row, col, value)
int row, col;           /*  location to store the value  */
double value;           /*  the actual value to store  */
{
        wvalue[row][col]=value;
        return (value);
}

/* source():  return the  W value referenced by input parameters  */


double source(row,col)
int row, col;           /*  location to look up */

{
        return (wvalue[row][col]);
```

```
}

/* solutn(): print solution step at console */
solutn(j,i)
int j,i;        /* element numbers  */
{
        printf("\nt=%7.3e, y%1d=%7.3e, y%1d_exact=%7.3e, diff=%7.3e",
                tarray[j],i,source(j,i),i,exact(i,tarray[j]),
                source(j,i) - exact(i,tarray[j]));
}

/* main program: */

main()
{
        /* external declarations */

        double store(), source();

        double fn();    /* ensure that this is typed as double */

        /* local variables: */

        register int i,j;


        double init[SYSIZE];            /* initial condition matrix */

        /* begin code: */

        printf("\n\nrkst2.c     as of 25-Jul-84\n\n");
        printf("        Integrates the differential equation system:\n\n");
        printf("        u1'(t) = 8u2(t)                 u1(0) = 10\n");
        printf("        u2'(t) = 2u1(t)                 u2(0) = 7\n");
        printf("        for which the exact solution is known to be:\n\n");
        printf("        u1(t) = 12exp(4t) - 2exp(-4t)\n");
        printf("        u2(t) =  6exp(4t) +  exp(-4t)\n\n");
        printf("        for     t = %7.3e to %7.3e, with %u steps\n\n",
                TSTART,TEND,STEPS);

        /*
                integrate the answer from t = 0 to t = 10 sec
                STEPS points.
        */

        init[0] = Y1ZERO;       /* set up initial condition matrix */
        init[1] = Y2ZERO;

        rk4n(fn,source,store,SYSIZE,TSTART,TEND,STEPS,init,tarray,yarray);
                        /* compute the answers for 1 function */

        /*  Print out solutions  */

        for(i=0;i<SYSIZE;i++)
                for(j=0;j<STEPS;j++)    /* print solution */
                        solutn(j,i);


        printf("\n\nEnd of execution\n\n");
}
```

End Listings

BOOK REVIEWS



CP/M Techniques

Ken Barbier
Published by Prentice-Hall, Inc.
225 pages
Reviewed by Dennis Cashton

If you are as much of a CP/M nut as I am, this book will become indispensable for you. In a very clear, consise, and friendly manner, Ken Barbier manages to discuss the real "guts" of CP/M. If you have worked with modifying your BIOS using assembly language under CP/M and have had to do anything that requires looking at the Digital Research documentation, your hair is probably lying in piles on the floor at the foot of your disk drives. This book takes the mystery out of all the vague topics about which "mother never told you."

I consider Mr. Barbier's previous work *CP/M Assembly Language Programming* a prerequisite for reading this book, but even if you have only a working knowledge of the mechanics of CP/M with an assembler, that should be good enough to get you through the book. In fact, even if you were a complete novice to CP/M, you'd just about be able to read and understand *CP/M Techniques*. (The "novice" level of CP/M is obtained by reading and not understanding the Digital Research documentation provided.)

The book starts with a general overview of a computer, what constitutes hardware, software, and firmware, and what constitutes an application program. Barbier follows this with a brief glance at the function of an operating system, and then jumps directly into CP/M. From this point on, the book is packed with sample programs and subroutines, each of which is immediately useful as well as being an excellent example of good programming practice. In fact, while reading the book, I had to run to the computer and enter each of the programs, just to make sure that they really do work. They do.

He goes on to a discussion of the medium of floppy disks. This section covers everything from history to current standards, and directions for the future (plus some more sample programs). Finally, he plunges head first into the part of CP/M that scares the most people: the BIOS. All the information and techniques illustrated in the prior chapters are brought to bear on the task of creating a customized BIOS for a user's own hardware configuration. What makes this section specially clear is the use of real-life examples. The book is full of them. They make such as abstract concepts like the IOBYTE much easier to understnd, mostly because you can see a real case of how and *why* each is used.

After you finish this book, it no longer seems like a monumental effort to get your CP/M to work with multiple printers, or different disk drives, or even a hard disk drive. You walk away from reading this book with a feeling that you can now conquer the (CP/M) world.

The book includes two appendices. The first is the ASCII code with mnemonics, keystrokes, hex, and function. The second is a listing of all the general purpose subroutines printed in the book, and a brief synopsis of their function.

In summary, if you have ever had a good mystery novel that you just couldn't put down, then you will understand the feeling you get from reading this book. It's so full of useful and concisely presented information that it is a pleasure to read. Even if you are not interested in customizing your own BIOS, this book is worth having for the programming techniques alone. If, on the other hand, you have the nerve to want to mess with your own hardware and BIOS, you'll find this book an invaluable tool.

The RS-232 Solution

Joe Campbell
Published by Sybex, Inc.
$16.95, 225 pages, illustrated, paperback
Reviewed by Dennis Cashton

Here's the plot: You get your new printer, the one that will do everything you ever wanted in a printer and more: high speed dot-matrix "correspondence quality" graphics with a 400K buffer. You assume that because the printer is "RS-232 compatible" you can plug it into the RS-232 port on the back of your computer, and everything will work fine.

Rushing home from the store with the printer, you tear the packing material off like it's wrapping on a box under the Christmas tree, glance at the cover of the instruction book, and lightly toss it aside. You don't need to look at it because "if it's RS-232, all I have to do is plug it in!" *Wrong!* You hook up the cable you bought with it (that the store owner said was a "standard" cable). If you're really lucky, the printer and the computer both have female connectors, because the cable you bought has male connectors on both ends. You turn on the computer. That part you've done a hundred times before, and you get no surprises. You turn on the printer, and the power light comes on, the printhead jumps a little, and the motor whirs. So far, so good.

You boot up CP/M and get no surprises there. With you heart in your throat, you hit control-p to toggle the printer on, and *zappo!* No more CP/M prompt! The keyboard is dead!!! What happened? Or better yet (more mystifying anyway) you get the prompt

back but you try to print something and *nothing* comes out. Is it the cable? The baud rate? Is the printer no good? What do you do now?

Well, this is a good time to look at the instructions book for the printer, the computer, and the operating system. Barring all complications, chances are the manufacturers of your printer and computer haven't totally lied about their RS-232 compatibility, but there are many ways to stretch the issue.

The RS-232 Solution will help anyone to connect any two RS-232 devices. It's the kind of reference book many of us need—there are, however, a few ommissions and discrepancies in this one. Campbell is careful to explain everything there is to know about RS-232, including all the buzzwords; unfortunately, there are some gaps in relating the buzzwords to the real signals and their meanings.

The author wisely sets up a convention in labeling his data-flow diagrams. He starts out writing the names of inputs in lowercase letters, marking them with a ?, and writing outputs in uppercase letters, marking them with a !. On the very next page (and several other times in the book), he violates that convention. Although a minor inconvenience, it illustrates the nature of some of the book's flaws. I got the feeling that the book needed a technical wizard as an editor.

In spite of this minor beef, this book contains a wealth of knowledge about interfacing RS-232 devices. A set of instructions is developed that, if followed, will allow any real RS-232 device to connect to any other. The book is structured well, and can take even the total novice through the logic and signals of the RS-232 standard.

In explaining each one of the signal lines, the author uses many illustrations that make it easy to understand who's doing the talking and who's doing the listening. He also uses (usually humorous) cartoons illustrating excellent analogies to the purpose and functions of the handshaking lines, making these abstract concepts clear. There is even a chapter explaining the UART and how it functions in a serial interface. The book discusses signaling buffer full, pausing the transmitter and receiver, and generally what handshaking is all about. There is even some mention of software handshaking, but since this book deals mostly with the hardware end of things, it is understandably brief.

Eventually the book plunges the reader, who is armed only with the knowledge and theories gained from the previous chapters, into the real world of finicky devices. Topics with hobbyist-sounding connotations like "tricking the interface" come up, but as the reader soon finds out, this sort of fiddling is necessary in the real world. At this point in the book, you get the facts of "what happens if I don't do this." Once you get through the next couple of chapters, you are set to try things out on a real device.

In fact, one chapter discusses a simple tool that will make it easier to start out. I'm sure that many people have considered buying one of the "breakout boxes" on the market that allows them to configure a cable that will make their printer (or modem) work with their computer. The author describes here how to build, at a savings of over $100, a simple tool that performs the same function. I made one out of "junk box" parts in a little under 15 minutes, but if you had to buy all the parts new, it would still cost less than $20.

After this, you get to put your hands on your devices, and let the sparks fly. Actually, Campbell is careful to point out what the RS-232 standard says on that subject. According to the standard, any interface line must be capable of withstanding a direct connection to any other line without damage to the interface or the equipment. So there is really no danger of sparks. With this knowledge firmly in hand, we march on to learn and practice the interfacing technique.

Steps are numbered 1 through 5 and must be completed in order. They are:

1) Set baud rate
2) Ascertain sex of the equipment
3) Satisfy device control logic
4) Locate the handshaking
5) Specify the cable

As we learn later, some of these steps get combined, but they are still done in this general order. Each one of the steps is explained in detail, and you may now go happily on your way, connecting DTE's to DCE's all day long.

A latter section in the book takes you step-by-step through five actual case studies. They range from the simplest possible to the very tricky and difficult, but each one serves as an excellent example of all that has come before.

Basically, in spite of some of the confusion due to inconsistencies and unconnected concepts, this book should be on the shelves and workbenches of anyone who deals with these "universal" RS-232 devices. You could get the information in this book elsewhere, but only with much effort. I personally expect the pages of my copy of this book to become dog-eared and dirty.

The IBM PC Connection

James W. Coffron
Sybex Computer Books, 1984
$16.96, 260 pages
Reviewed by Eunice B. Ordman

The *IBM PC Connection* is a book for those with an elementary knowledge of BASIC (the IBM version of Microsoft BASIC) and no knowledge of electronics. It is a book for hobbyists, not a textbook. The author takes the reader step-by-step through both software and hardware aspects of interfacing the IBM PC. The uninitiated should be able to follow the author's instructions but may not understand much of what he is doing, particularly at first—Coffron tends to explain the functions of most of the components, without explaining how they work.

Readers will learn how to do sophisticated things such as building an alarm system that tells which switch set off the alarm. Other topics include computer speech output, temperature measurement with the computer using an integrated circuit as input, computer control of a variable speed motor, and analog-to-digital and digital-to-analog conversion.

For the uninitiated, Coffron discusses a preassembled input/output board and then progressing to more complex topics, he gives diagrams of circuits the reader can build. He does not present the reader with a huge complicated circuit, but introduces the parts of the circuit gradually—it takes perhaps four diagrams to present a given circuit. He even includes tips on

reading schematic diagrams. The reader is on his own when it comes to soldering, wire wrapping, and so forth, but such information is widely available. The appendix gives the necessary manufacturer's specifications for each integrated circuit used and the name of a supplier for each type of part used. (The suppliers on his list whom I've used have very reasonable prices.)

Coffron tells how to revise the circuits and programs for different circumstances. With a few more sentences, he could have given an explanation of address decoding, but instead, Coffron contents himself with showing circuits that use certain port addresses, using those addresses in his programs. He gives a diagram of the IBM PC Input/Output Connector, showing the abbreviations for all 62 signals, but he does not tell what many of the abbreviations stand for.

I think the average hobbyist will be glad that the author did not confuse him with unnecessary details. I myself would have preferred a bit more explanation, however.

Advanced Pascal Programming Techniques

Paul A. Sand
Osborne/McGraw Hill, 1984
$14.95, 370 pages
Reviewed by Bob Langevin

If L-Name is the name of a computer language, your favorite bookseller's shelves are filled with books with titles such as *Introduction to L-Name*, *Easy L-Name*, and *L-Name for Beginners*. When, you wonder, will you get to the really advanced stuff? Sand's new book, *Advanced Pascal Programming Techniques* (APPT), is one of the few available Pascal texts that will take you well past the introductory level.

Overview

Sands correctly observes that most introductions to a computer language provide numerous small examples to illustrate each of the language's features but few texts, if any, show how these features are used in building substantial and meaningful programs. His book is a serious attempt to remedy this defect.

In its introductory chapter the book discusses "What is a Good Program"; the remaining seven chapters of the book are a series of detailed case studies, each illustrating important design concepts. The chapter titles give a good indication of the material covered: "CRT Techniques"; "Interactive Input"; "Crunching Numbers—A General Purpose Calculator"; "Text File Tools"; "Games and Strategy"; "Simulation and Animation"; "The Plane Truth—An Electronic Worksheet".

Because the Apple II version of UCSD Pascal was used to develop the programs in the book, the appendix is especially useful—it discusses the problems of program portability and shows how features unique to UCSD Pascal and the Apple II can be implemented in the context of other Pascal compilers on other machines.

Features

The first chapter discusses the features that make a good program, viewed from both the user's and the programmer's point of view, an important distinction that is often not made. From the user's viewpoint, the important features are usefulness, ease of use, efficiency, flexibility, reliability, and suitability; from the programmer's, they are readability, portability, clarity, and modularity. Sand emphasizes the need to recognize that, during its lifetime, a real program will be repeatedly fixed, modified, and enhanced—often by someone who did not design or implement the original. The succeeding chapters show, by example, how the user's needs can be met and the programmer's objectives achieved.

It is noteworthy that the design of the applications in each of the case studies has been carefully crafted so that procedures and functions developed in one chapter become useful tools in the design and implementation of the case studies in succeeding chapters. There is a definite flavor of the Unix design and development philosophy here.

In the second chapter, on CRT techniques, Sand shows how to write a Pascal procedure to manage the cursor and screen control features of a CRT. These are used in implementing a simple program that creates, displays, and solves mazes. This program is of little inherent interest, and the brief discussion of CRT management procedures in the beginning of the chapter could have better appeared as an introduction to the next chapter of interactive input.

Unquestionably, the effective management of user interaction with a program is of critical importance in achieving user satisfaction and controlling input errors of all kinds. Sand, in Chapter 3, provides a good discussion of interactive input and error control in the course of developing "gaslog," an application program that maintains and reports gasoline usage data for an automobile. Surprisingly, this deceptively simple program provides the motivation for developing procedures for string input and string manipulation, boolean and fixed-point numeric input, date handling, and a variety of type transformers to convert between string and various numeric types.

In Chapter 4 Sand uses the development of a general-purpose calculator as a vehicle for addressing error control, a data structure for "extended" real numbers, a simple parser of user input, and functions to convert back and forth between strings and the "extended" reals that Sand has devised.

Next Sand discusses text file tool. These include tools to open or close an existing text file to read a string from or write a string to a text file, to create a new text file for output, and to get a text filename from the user. These functions are used to develop a simple file copy program and, much more interestingly, a rather elegant program for printing text files. The latter offers the opportunity for a good discussion of the use of escape sequences for initializing a printer and the incorporation of procedures for changing print parameters to control the format of the output.

The print program also supports chaining of files to be printed. Aside from its utility as a learning device, this print program is a useful tool for any program developer. In addition if used with a good full-screen text editor, the combination makes a serviceable word-processing package.

Chapter 6 is devoted entirely to the design and implementation of a program to play Reversi—with a brief side trip to describe Apple II graphics

capabilities. While this has some interest if you happen to want a program to play Reversi, the design considerations are so unique to the structure of this particular game and game board that you can neither generalize nor extend considerations of Reversi's appropriate data structures and procedures for moving pieces and position evaluation to other games. If you are not a Reversi fan, you can skip this chapter without losing much.

Next, simulation and animation are illustrated by designing and implementing two simulations: "bounder," which simulates the motion of several balls moving inside a box under the influence of a constant acceleration field, and "isaac," which uses many of the same program components and simulates the motion of several balls moving under their mutual gravitational attraction. Several sets of initial conditions are provided for "isaac" that result in simulations of earth-sun, earth-moon-sun, and Lagrangian systems. The displays associated with both simulations depend entirely on the graphics display capabilities of the Apple II and would require major revision to accommodate the graphic characteristics of other computer systems.

Sand's book concludes with the development of "pascalc," a basic but usable spreadsheet program. Although this program does not approach the sophisticated capabilities of commercially available spreadsheet programs, it affords a useful study of Pascal procedures for dynamic memory management. In addition, and since any useful spreadsheet has dimensions substantially larger than the available screen area, "pascalc" includes the development of effective screen management procedures. As in the preceding chapters, Sand has taken some care to design "pascalc" so that it makes substantial use of procedures and functions developed earlier in the book.

The book is strengthened by the inclusion of suggestions and recommended reading at the end of each chapter. The suggestions, some of which are quite challenging (and time-consuming), address possible and desirable enhancements to each chapter's program. The references included in the recommended reading are usefully annotated to indicate their relevance to the text—a uniformly desirable procedure.

Evaluation

There is no doubt that you can improve prove your programming skills by careful study of Sand's book, especially if you actually work through in detail the sections of interest to you. His book failed, however, to meet my expectations in several important respects.

First of all, the book is too "even." Sand gives you no indication whatsoever of the relative importance or general utility of any of the ideas, procedures, or functions discussed. A twelve-line procedure to send a termination sequence to a printer, for example, receives neither more nor less emphasis than "stor," which converts a string into a real number.

Second, because the book is structured around explicit case studies, related topics end up scattered through several chapters. By way of example, various functions to convert between strings and numeric data types appear in chapters 3, 4, 6, 8, and 9. As a result, the basic characteristics of such type conversions are never developed and presented. Data structures likewise "pop up" when they are required, so that an organized view of data structures and their relationships is entirely absent.

Finally and again due to its structure, the book isn't very suitable for reference purposes—you can't really look anything up in it. As indicated earlier, the table of contents is only eleven lines long, so it isn't of much help. The index isn't much better, particularly because most index entries reference data structures, procedures, and functions that are unique to the book's case studies. When the entries do purport to be more general, they are often nearly useless; for example, under the major heading of Algorithms the only subheadings that appear are Minimax and Shell sort!

In summary, *Advanced Pascal Programming Techniques* does develop numerous useful tools that are applied to the design and implementation of substantial programs. It does not, however, live up to its title. Techniques, as such, are never really discussed. You won't find in the book an organized presentation of typical "technique" topics such as screen design, error trapping, algorithm efficiency, program efficiency, data structures, modularity, design for maintainability, program testing, and documentation. If you prefer a strong technique orientation, you would do much better with a book such as *Advanced Programming and Problem Solving with Pascal* (Schneider and Bruell, John Wiley & Sons, 1981)—in my opinion, the best text available on advanced programming techniques.

Computer Algebra

V.162 of Lecture Notes in Computer Science

Edited by J. A. van Hulzen
Springer-Verlag
305 pages
Reviewed by Morton F. Kaplon

This volume, No. 162 in the Springer-Verlag series *Lecture Notes in Computer Science*, is the third one dedicated to a computer algebra conference. The conference, EUROCAL '83, was held at the Kingston Polytechnic, Surrey, England, from 28 March, 1983 to 30 March, 1983. EUROCAL represents the EUROpean Computer ALgebra Community. The conference was organized under the responsibility of SAME (Symbolic and Algebraic Manipulation in Europe) and in cooperation with SIGSAM (Special Interest Group on Symbolic and Algebraic Manipulation) and with the official approval of ACM.

A natural question is "What is computer algebra?" A response is best given by quoting from the concluding statements of Professor van Hulzen's introduction:

"Many of the conference participants recognized that publication of real lecture notes about fundamental aspects and use of computer algebra, for instance in this Series, might largely contribute in establishing the user community and the more general interest computer algebra deserves, at least according to its adepts. This will certainly contribute to a *communis opinio*, and probably also to a 'definition'."

Does that leave you puzzled? This volume presents 27 research papers,

organized in seven categories as follows: Algorithms 1 — Miscellaneous; Applications — Miscellaneous; Systems and Language Features; Algorithms 2 — Polynomial Ideal Bases; Algorithms 3 — Computational Number Theory; Algorithms 4 — Factorization; and System Oriented Applications. This volume and the papers presented are not for the novice and do not constitude easy reading. The range of material covered is quite broad, covering the spectrum from aspects of pure mathematics to the realization of programs in terms of specific hardware.

The first paper, entitled "Integration—What do We Want from the Theory?", reflects one end of that spectrum. The author, in discussing the state of the theory relating to the integration of algebraic functions, proves an existence theorem using a non-constructive proof—certainly, not a very useful result for those interested in writing programs. At the other end we have "Implementing REDUCE on a Microcomputer" (this is not a recipe) and "The Design of MAPLE: A Compact, Portable, and Powerful Computer Algebra System." The latter will yield to the programmatically inclined a sense of the aims of *Computer Algebra* as reflected in software.

Do not be misled, however. As noted, the range of *Computer Algebra* is broad, but that range does include significant aspects relating to artificial intelligence. There also are articles that are certainly germane to developments in microcomputer software and perhaps even to hardware at a very basic level. These include, among others, "A Knowledge-Based Approach to User-Friendliness in Symbolic Computing," "Computer Algebra and VLSI," and "The Bath Concurrent LISP Machine." If you want to get a feel for what is going on at the frontier of this interesting and important field, a few dedicated hours of selective reading from this volume may prove quite informative and even potentially useful. And if you know LISP, you will even occasionally feel at home.

New Books

In addition to books we formally review, we see lots of titles that we think you might as least want to know about. We will from time to time provide a list of such books along with a brief impression or description. Here is our first batch.

Ada—An Advanced Introduction Including Reference Manual for the Ada Programming Language
Narain Gehani
Prentice-Hall, Englewood Cliffs,New Jersey, 1984.
$19.95, 291 pages
One thick book.

Advanced Programming in Microsoft BASIC
Gabriel Cuellar
Reston Publishing, Reston, Virginia, 1984.
$16.95, 152 pages
A "second" book on BASIC. Contains programming tools and hints.

Directory of Public Domain (And User-Supported) Software For the IBM Personal Computer
PC SIG
PC SIG, Santa Clara, California, 1984.
$4.95, 109 pages
The PC Software Interest Group's directory.

Free Software For the IBM PC
Bertram Gader and Manuel V. Nodar
Warner, New York, 1984.
$8.95, 462 pages
Software on 45 electronic bulletin boards.

Interactive Programming Environments
David R. Barstow, Howard E. Shrobe, and Erik Sandewall, eds.
McGraw-Hill, New York, 1984.
$34.95, 570 pages
Adele Goldberg on Object languages, Brian Kernighan on Unix, Terry Winograd looking beyond programming languages.

Invitation to Ada
Condensed Version
Harry Katzan, Jr.
$14.95, 166 pages
Petrocelli, New York, 1984.
Hospitable Harry Katzan, Jr., has written a series of "Invitiation to—" books for Petrocelli. Here, he invites the almost-beginner to sample the rudiments of the language they speak at the Department of Defense.

Learning LISP
Jeff Shrager, Steve Bagley, Steware Schiffman and Steve Cherry
Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
$14.95, 199 pages
Comes with a LISP disk for the Apple.

On Conceptual Modeling
Perspectives from Artificial Intelligence, Databases, and Programming Languages
Topics in Information Systems series
Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, eds.
Springer-Verlag, New York, 1984.
$29.25, 460 pages
How to encode knowledge. Papers presenting three approaches to a fundamental problem in the design of the commercial artificial intelligence packages called expert systems.

Pascal For Programmers
Olivier Lecarme and Jean-Louis Nebut
McGraw-Hill, New York, 1984.
$22.95, 267 pages
Presents ISO Standard Pascal. Assumes knowledge of at least one high-level language. The authors believe that Pascal enjoys the popularity it does partly because, when it was released, no agency, organization or vendor made any attempt to support it.

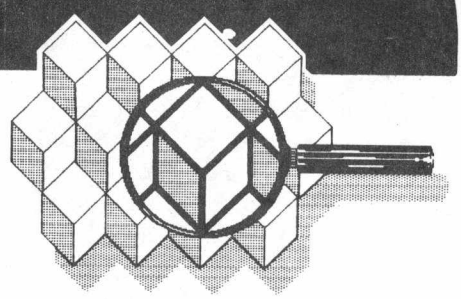
Programming In C
Stephen G. Kochan
Hayden, Hasbrouck Heights NJ, 1983.
$18.95, 365 pages
Attempts to speak both to complete novices and to experienced programmers.

Programming With Structured Flowcharts
Krishna K. Agarwal
Petrocelli, New York, 1984.
$12.00, 166 pages
Why you should use those enclosed-space Nassi-Schneiderman flowcharts.

The Elements of Friendly Software Design
Paul Heckel
Warner, New York, 1984.
$8.95, 192 pages
Serialized in *InfoWorld* in 1982. How to program like D. W. Griffith. Heckel is nothing if not eclectic.

(Continued on page 127)

OF INTEREST



by R.P. Sutherland

Software Tools

Borland (Turbo Pascal) International is now offering **Turbo Toolbox** to complement its Pascal compiler for Z80 and 8088/8086 microcomputers. Turbo Toolbox is a set of programming tools for data base, terminal installation, and sort applications. Two of the five tools, Turbo-ISAM (Index Sequential Access Method) and Quicksort, are available with commented source code. Turbo Toolbox is available for $49.95 from Borland International, 4113 Scotts Valley Drive, Scotts Valley, CA 95066.

Csharp Realtime Toolkit from the Systems Guild is a set of real-time, multitasking C programmer tools distributed in source code to allow user modification. Cgraph, for example, lets programmers write portable graphics programs and configure graphic system parameters by using C procedure calls. Csched (real-time scheduling of user procedures) creates a multitasking environment where each scheduled procedure can turn to completion unless interrupted by a procedure with a higher authority. Csharp tools are processor independent and run on 8- and 16-bit processors as well as on the entire PDP-11 family. Csharp tools will run under many operating systems, including Unix and RT11, and can also be imbedded in stand-alone software. The Csharp Realtime Toolkit is available under a single source license for $600 from Systems Guild, Inc., P.O. Box 1085, Cambridge, MA 02142.

C_to_dBASE, from Computer Innovations, provides the ability to write C applications for dBASE files. The $150 price tag (there are no additional royalty charges) includes the complete source code plus 70 functions that enable C programmers to perform operations on dBASE and index files. C_to_dBASE is available directly from Computer Innovations, Inc., 980 Shrewsbury Avenue, Suite R, Tinton Falls, NJ 07724.

iLISP, from Computing Insights, is a new implementation of LISP for Z80 microcomputers. iLISP runs under CP/M 2.2 and is based on the LISP dialect developed by Gerry Sussman called SCHEME. One of the advanced LISP features includes a complete, extendable implementation of ELIZA, the famous psychotherapist parody. iLISP is available on both 8- and 5¼-inch disk formats (including Kaypro, Morrow, Zenith, and Osborne). The list price of $49.95 includes a 60-page introduction to iLISP programming. For more information write: Computing Insights, P.O. Box 4033, Madison, WI 53711.

Graphics

The **STB Graphix Plus II** is a video adaptor board for the IBM PC that supports both color and monochrome displays. I dropped one into my Eagle PC and now my Eagle flies "Flight Simulator." Three interesting extras are: (1) a 64K printer buffer; (2) "PC Accelerator," which supports print spooling and which offers a quick-start option that fools the PC into thinking only 64K of RAM is resident; and (3) an electronic RAM disk utility program. The package includes a 16-color driver for Lotus 1-2-3. The price is $495. For additional information contact STB Systems, Inc., 601 North Glenville Ave., Suite 125, Richardson, TX 75081.

Voices

ProTalker by Speech Ltd. provides S-100 systems and IBM PCs with voice output capability. Advanced users can interface ProTalker to most programming languages and applications because source code is provided. ProTalker is a digitizer/synthesizer that is switch selectable to rates of 4, 6, or 8 kHz. Adaptive delta pulse code modulation is used to reduce the size of digital recordings. Recordings are stored on disk until needed and can be accessed randomly under program control for play back. A telephone demo of ProTalker is running at (415) 858-2795. The price is under $350, available from Speech Ltd., 3790 El Camino Real, Suite 213, Palo Alto, CA 94306.

SynPhonix 100 is a speech synthesizer for the Apple II family. The board plugs directly into Apple II slots 1 – 7. The package includes a speech operating system on diskette. Users can generate speech and sound effects and incorporate them into software with standard BASIC statements. The SynPhonix 100 retails for $135 and is available from Arctic Technologies, 2234 Star Court, Auburn Heights, MI 48057.

The **VocaLink Speech Recognition Board** is a single-slot voice recognition board and software package for the IBM PC that allows users to operate off-the-shelf programs with up to 240 spoken commands. The SRB incorporates a high-speed, 16-bit Intel 80186 microprocessor to manage the complex operations required to accurately recognize discrete words/phrases spoken by a specific individual. Other hardware features include: the ASA-16 (a custom audio spectrum analysis

chip), 128K RAM, 32K EPROM, and the cabling required to connect a microphone, headset earphone, and voice synthesizer. The VocaLink Speech Recognition Board costs $1700 and is aimed at OEMs and sophisticated end-users. Inquiries should be addressed to Jim Bright, Interstate Voice Products, 1849 W. Sequoia Ave., Orange, CA 92668.

Macintosh

MacModula

Modula Corporation believes that, compared to Modula-2, using any other programming method is analogous to doing arithmetic in Roman numerals. If Modula-2 is to replace Pascal, then a Pascal-to-Modula-2 converter seems a sensible product. Modula Corporation has announced a range of software tools for use by developers of Modula-2 applications. A Modula-2 compiler, interpreter, and a **Pascal-to-Modula-2 source-code converter** are available for Macintosh as well as for Apple's other machines (and IBM PCs and compatibles). The Macintosh versions feature enhanced, bit-mapped graphics support, which the company claims is superior to any Apple-supplied Macintosh products. Modula Corporation is at 950 N. University Ave., Provo, UT 84604.

The Ultimate Mr. Potato-Head

MicronEye for Macintosh is an imaging peripheral that allows the Macintosh to take pictures! The user can accept an image into MacPaint and then subject that image to all of the features of MacPaint. The possibilities for Mr. Potato-Head-like manipulation of one's friends or enemies are staggering. MicronEye is available for $395 from Micron Technology, Inc., Vision Systems Group, 2805 E. Columbia Road, Boise, ID 83706.

Mac Learns to Read

Oberon International has a Z80-based type-reader that is compatible with Macintosh. **Omni-Reader** can read four different typefaces and has the ability to learn others. The device allows one to read text into a computer at the rate of 80 characters per second. Omni-Reader is manufactured in the United Kingdom and retails for $500. Contact Oberon International, McArthur Plaza, Suite 630, LB48, 5525 McArthur Blvd, Irving, TX 75062.

Attention User Groups

If you would like your user group to be included in a ***National Directory of User Groups***, send a stamped, self addressed envelope to: Ken Ryder, P.O. Box 4102, Rome, NY 13440.

BLAISE PASCAL (1623-1662)

Chips Off the Old Block

Kids are fast unlearning passive (television addict) approaches to cathode ray tubes and discovering that monitors connected to microprocessors are devices to make dance.

Computers 'n Kids Adventure Fair is looking for games and educational software written by people up to 17 years old. Winners will have the chance to demonstrate their programs at the first annual Computers 'n Kids Adventure Fair, which will be held at the San Francisco Concourse (8th and Brannan) from October 18 – 21. For more information call (415) 848-6860.

Isaac Asimov has published a book for children called ***How Did We Find Out About Computers?*** Asimov places the invention of the computer in the context of history by tracing the development of man's ability to calculate from counting on fingers, to the abacus, to the slide rule, to the inventions of Pascal and Babbage. The illustrations by David Wool include Pascal (illustration center page), Jacquard's first loom, the Hollerith tabulator, Vannevar Bush's differential analyzer, and Aiken's Mark One. Asimov concludes this well-written and informative history with an affirmation of human intelligence (or is it a challenge to *DDJ* readers?): ". . . when I write a story, I write as fast as I can and put one word after another in just the right order until I am finished. But how can I tell a computer to do it? Even if I put a whole dictionary of words into its memory, how can I tell it what word to put first, and what second, and what third? How can I explain to it how to choose the order of words so that it can write a brand-new story, just the way I do it, when I don't know how I do it?" *How Did We Find Out About Computers?* is priced at $8.85 (ISBN 8027-6533-5) from the publisher, Walker and Company, 720 Fifth Avenue, New York, NY 10019.

DDJ

(Continued from page 124)

The IBM PC-DOS Handbook

Richard Allen King
Sybex, Berkeley, 1983.
$16.95, 288 pages

Includes appendices on the differences between PCDOS and MSDOS and among early version of PCDOS.

DDJ

A Guide to C Resources

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

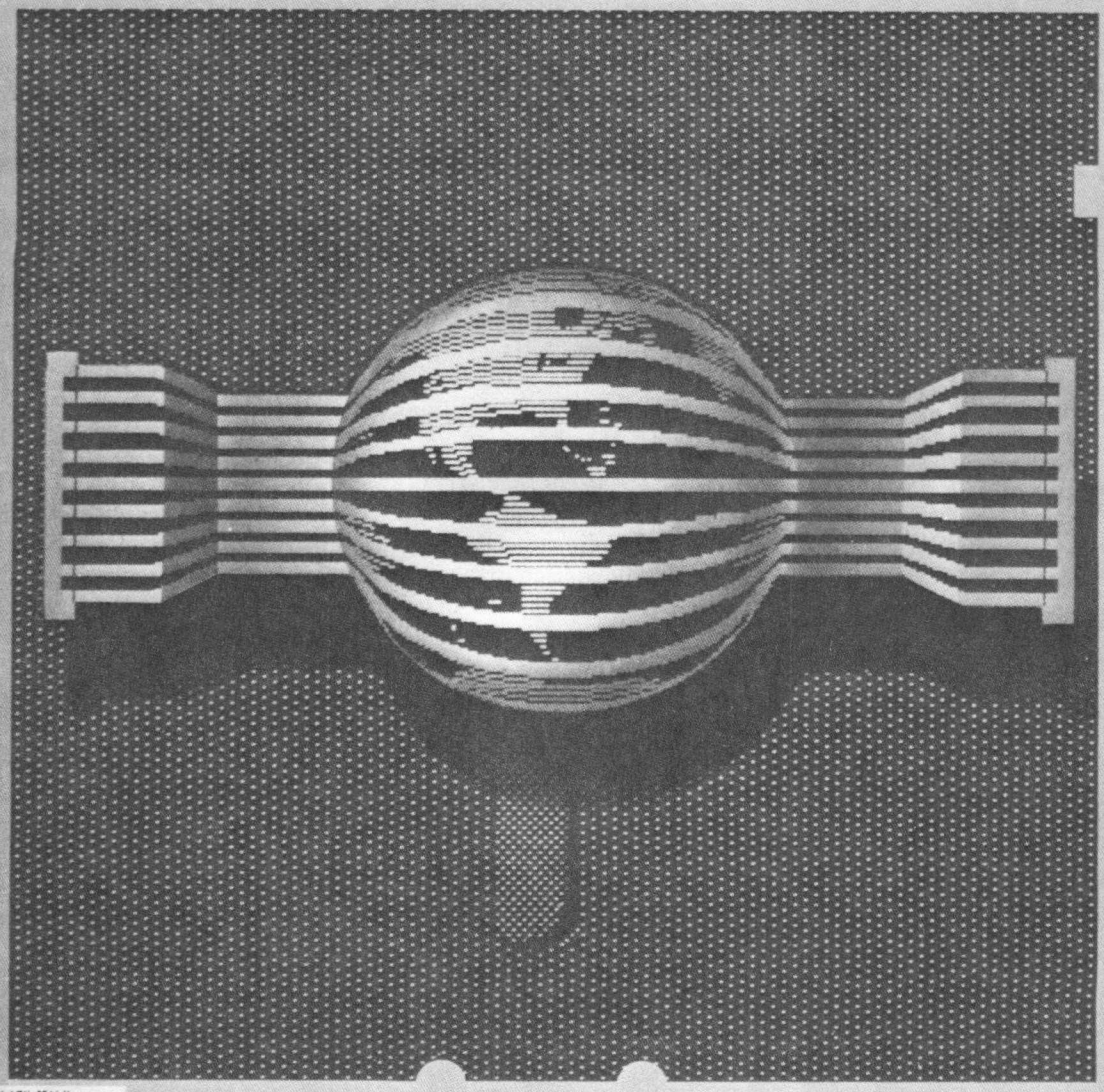
#97 November 1984

$2.95 (3.50 Canada)

Program Monitor Package

CP/M 2.2 Goes PUBlic

New Column: Computer Calisthenics



0 38351 16562 11

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*
Editorial Intern *Mark Johnson*

Production
Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover *West and Moravec*

Advertising
Advertising Director *Stephen Friedman*
Advertising Sales *Walter Andrzejewski, Shawn Horst, Beth Dudas*
Advertising Coordinators *Alison Milne, Lisa Boudreau*

Circulation
Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

November 1984
Volume 9, Issue 11

CONTENTS

In This Issue

As promised, Michael Wiesenberg starts his column of puzzles for computer solution this month. In addition, we tell you about modifying proprietary software, circumventing the boundaries of user areas in CP/M, determining where your program is spending most of its time, and using a binary tree to maintain a spelling dictionary. There is even a token parsing filter for MSDOS.

Another very useful item is the guide to C programming resources. Until we started working with Terry Ward to get his resource list into production we had little idea how much work such a project can be. As the list grows, maintenance can become as cumbersome as the initial compilation: addresses and product lines change, new companies enter the picture, old companies give up the ghost, new material is published each month . . . .

Eventually you just have to decide to stop updating and print the thing. We did succumb, however, to the temptation of last minute additions to the bibliography. While it was too time consuming to do a complete update, we did try to include as much of the latest material from *Dr. Dobb's* as we could. We hope that folks will understand our selectivity in this matter.

Finally, it is almost inevitable that sources will be omitted or come into being after you quit updating. If there are folks who did not get included, drop us a note. If we get enough new information, perhaps we can convince the author to do an addendum.

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to remarks to the editors concerning accuracy and relevance of manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness. Their remarks help prevent authors from exposing blindspots or misconceptions in print and help ensure that our readers receive clear and accurate information.

Each month, we print the names of the referees who contributed their insights on material in that particular issue. This space-conserving scheme, unfortunately, does not mention those who work on material that doesn't make it into the magazine. We are in the process of updating our records right now, and will publish another complete list of the board shortly.

The referees who contributed to this month's issue are:

Ian Ashdown, P.E., byHeart Software
Robert Blum, *DDJ* Contributing Editor
David D. Clark, Pennsylvania State University
Michael P. Kelly, Design Software
Douglas W. Rosenberg, Optimal Software
Stan Sieler, Next Generation Systems
Allen Tigert, Krontron Medical Electronics

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

Adding Primitive I/O Functions to muLISP 18
by Michael Carter
How to add new functions to muLISP, with a look at issues involved in modifying any piece of proprietary software.

Program Monitor Package—Using Interrupts to Instrument Applications 26
by Alan Bomberger
Optimize intelligently by monitoring where your program is spending most of its time.

CP/M 2.2 Goes PUBlic 48
by Bridger Mitchell and Derek McKay
How to make any file accessible from any user area on a given drive without the duplication necessary to many other schemes.

A Guide to Resources for the C Programmer 74
by Terry A. Ward
Including a bibliography and lists of program and product sources, this resource guide can help you start tackling the material available.

RESORT 88
by Donald G. Krantz
This program reads through a document after spelling corrections and uses a binary tree to add words to the user dictionary.

DEPARTMENTS

Editorial 6

Letters 8

Dr. Dobb's Clinic 12
The Sky is Falling, New APL Standard, MacBug, and more

CP/M Exchange 14
Programmer's Tool Chest Improved, CP/M v2.2 PIP Patch

Software Reviews 94
Windows for C v2.00, Mychess

16-Bit Software Toolbox 98
The IBM PC/AT, Counting Cycles, a programming pearl, a token parsing filter, and more

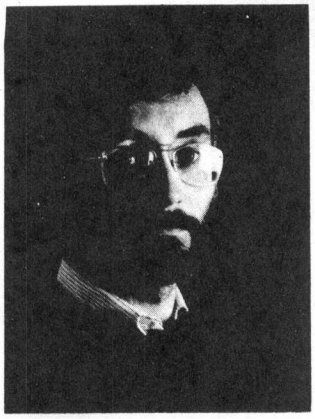
The Software Designer 114
A Realizable Fantasy: What programming tools would folks like to see?

Computer Calisthenics 116
A new column of puzzles for computer solution

Book Reviews 119

Of Interest 126

EDITORIAL



Computer magazines are dropping like dry leaves in a storm. This summer saw the fall of *Personal Software*, *List*, *Computers and Peripherals*, *Color Computer* and the entire Softalk branch: *Softalk, Softalk for the IBM Personal Computer*, *St. Mac* and earlier, *St. Game*. Then as the season changed came the announcements that *Microcomputing* and *Microsystems* would cease publication.

I am especially sorry to see those two drop off. Both magazines call up for me an electrically-charged time, days spent playing Little Brick Out on a cassette-based Apple and nights struggling to get a payroll system up on the Board of Health's multiuser Alpha Micro. I grew up in the Midwest, tornado country, and I recall one stormy autumn twenty years ago when my father stopped the car to stand by the side of the road, leaves whirling around his feet, and take pictures of the approaching funnel cloud. The memory of that day came back years later when, between sessions of Little Brick Out, I read early issues of those two magazines and felt another kind of storm brewing: I read in an ionized atmosphere.

Dr. Dobb's advertising sales people see the demise of *Microcomputing* and *Microsystems* as two competitors less. But what do they know.

Microcomputing began with pique and a poke at *Byte* magazine. Born in the late 1970s when the few existing microcomputer magazines were read exclusively by hobbyists, *Microcomputing* had a curious close connection with *Byte*, but to call them sister publications would be seriously to misunderstand the relationship. When *Byte* publishers Wayne and Virginia Green parted company professionally, Virginia took *Byte* and Wayne took it hard. He immediately threw himself into the creation of another computer magazine, which he called *Kilobyte*. What he hoped to do to *Byte* was perhaps a little too obvious in the title; he met with legal objections to its use, and compromised on *Kilobaud*. But however pique-provoked the magazine may have been, its genius was Wayne Green's passion for gadgets, and it spoke to its readers with a freshness and relevance and passion that perhaps only hobbyist magazines can attain. Over the years, *Kilobaud* evolved by way of *Kilobaud Microcomputing* to *Microcomputing*, spinning off a little family group of computer magazines along the way. In 1983, Wayne Green's magazine family was bought by CW Communications, Inc., a huge publisher of computer magazines, and this fall CW decided to retire *Microcomputing*.

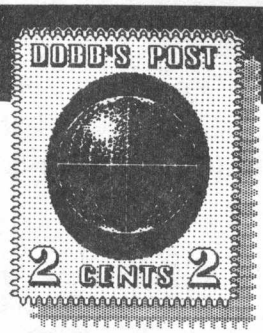
Microsystems was originally *S-100 Microsystems*, and was the vehicle through which Sol Libes spread his knowledge about S-100 bus computer systems and products and issues. Libes, even more than Green, was interested in spreading technical information to the community of hackers and tinkerers who evolved with the hardware into professional software developers and system designers. His magazine was always aimed at the advanced user. Eventually it evolved into *Microsystems* and was bought by Ziff-Davis, a major publishing house. Libes was still doing his popular "News and Views" column when, this fall, Ziff-Davis decided that *Microsystems* had no place in its lineup.

It's not that these magazines were perfect or that their publishers were wrong in retiring them. It's just that *Microsystems* and *Microcomputing* were *there*. They lived through, and helped to guide some of us through, a time when the world was changing magically, but the change was still so small you could hold it in your hand; a bright summer of computer hobbydom that is over now. Since I left the Midwest for Silicon Valley I haven't seen dramatic changes of seasons. Except for this one.

Michael Swaine

Michael Swaine

LETTERS



On Sixth Generation Computers

Comments on Richard Grigonis' "Sixth Generation Computers" article (DDJ, May 1984) ranged from laudatory to caustic. Michael Doherty's reply to Grigonis the following month drew further comment. Below is a sampling of some of the remarks we received. — Ed.

Grigonis' "Sixth Generation Computers" was stimulating, readable—the best article I've read in a long time!

Robert M. Mason
MRC—130 West Wieuca Rd, Suite 200
Atlanta, GA 30342

Regarding "Sixth Generation Computers," faster than light travel does not necessarily violate causality. Refer to "Is Faster Than Light Travel Causally Possible?" by Paul Birch, *Journal of the British Interplanetary Society*, v 37, pp. 117 – 123, 1984.

Peter G. Backes
24016 Kittridge St.
Canoga Park, CA 91307

"Sixth Generation" is too speculative, but interesting. Sixth generation computers will not use methods described. The human brain achieves intelligence without any of these "breakthroughs."

Kjeld Hvatum
P.O. Box 267, MIT Branch
Cambridge, MA 02139

In appreciation of the wonderful sketch of sixth-generation computers by Richard Grigonis (*DDJ* No. 91) I would like to share with your readers an essay by Miss Bonnie McIntosh:

Since the brain's synaptic signals are much like electronic impulses, a "brain modem" should be developed which could turn my thoughts into print and graphics on the computer screen. By utilizing this method of communication, thoughts would not be lost or altered by the slow process of verbalizing, drawing, or writing but immediately appear in pure form on the computer screen virtually as fast as they are thought—then edited as desired (a thought processing program?). The brain, through a brain modem, could become a high-speed input device. This process could also be reversed so that individuals could receive information directly from the computer to the brain, a method that could be much quicker than reading or listening to the information. It should also be possible to create or compose music in the same manner, so that the tunes that play "in my head" can be heard by others through brain-to-computer translation.

Peter C. Lincoln, Ph.D.
Bonnie S. McIntosh
Chaminade University of Honolulu
3240 Waialae Avenue
Honolulu, HI 96816

I enjoyed Michael Doherty's argument very much. We human beings are adaptive for survival, self defense, or greed. Machines, including computers, in the sixth generation still won't possess this inherent adaptive capability.

Dan Ley
84–15 168th Place
Jamaica, NY 11432

Comments to Grigonis' article may be forthcoming, but according to his interpretation of quantum mechanics he should already have access to my comments!

Dr. Jerry W. Lewis
555 E. Boyd Drive, Apt.C
Baton Rouge, LA 70808

This letter is in response to the Grigonis article "Sixth Generation Computers." It is really a clarification of a point.

As it is that only particles that travel faster than the phase-velocity of light cause the so-called Cherenkov Radiation and that the imaginary tachyons travel faster than the honest-to-god velocity of light, it must be that they do not radiate the aforementioned radiation but must most probably suck in radiation as they travel and as such could be said to emit black-body radiation, which leads us recursively to Max Planck's original theories that were the basis for quantum mechanics; as such, tachyons could be shown to be primitive recursive and therefore could be incorporated in today's computers via the so-called Ackermann function described in the same edition of *DDJ*.

Thomas Kellar
3104 Hassler St.
Dayton, Ohio 65420

Get rid of Grigonis.

Samuel Hahn
20800 Homestead Rd.
Cupertino, CA 94305

Along the Same Lines . . .

Dear Editor,

Your plaintive praise for the Macintosh user interface (August 1984 editorial) prompts these thoughts. In the future, eyeball-pointing sensors (already mounted on fighter pilot helmets) and voice recognition could provide effortless control of the cursor and word processing functions, eliminating the need for a third hand for the mouse.

The ultimate convenience in user interface may be control by thought, as suggested by the 30-year-old science fiction classic "Forbidden Planet" (MGM, 1956). In this movie, based on Shakespeare's "The Tempest," a central computer with control of immense power generators and matter creation machines is responsive to the thoughts of the planet's inhabitants. Want breakfast? Just think about food and it appears on the kitchen table. They perished when the computer created the monsters of their subconscious.

Sincerely,
Robert C. Briggs
1337 Rossway Ct.
Los Altos, CA 94022

A Couple of Corrections

Dear Editor:
While keying in Part II of "A New Library for Small-C" (June 1984, *DDJ* No. 92), I encountered a syntax error on page 60. In the ITOD.C function library on line 22, there is a missing semicolon.
The line reads
while(sz > 0) str[- -sz]= ' '
and should read
while(sz > 0) str[- -sz]= ' ';
I've not finished keying the whole library but so far this is the only error I've encountered.

Keep up the good work. I look forward to many months of association with your publication.

Sincerely,
Ken Brayton
9702 Ivanho St.
Spring Valley, CA 92077

[*Investigation revealed the most likely culprit to be a runaway knife in the production department. We'll try to keep them under better control. — Ed.*]

Dear Editor:
I congratulate you on a most interesting August issue. In reply to the article "What's The Diff" by D. E. Cortesi, please print the enclosed information for your readers. There is a fundamental flaw in the implementation, which I have corrected [see the listing on page 10].

The original program will fail in "pass5" if a block move includes lines that have been affected by "pass4." In fact, whenever a line is duplicated, as happens quite often in structured programming and in Pascal (e.g., begin, end, repeat), the symbol table pointer to any but the last duplicate line is lost. The required fix is to do the block moves (pass5) before applying Heckel's Rule 2 (pass 4a and 4b).

The author correctly identifies a very inefficient loop at the end of the procedure "resolve." His comment is that an additional pointer must be retained, at a cost of memory. I have changed the

Letters Listing (Text begins on page 8)

```
 1:
 2: (***************** EXCERPT FROM DIFF.PAS ************************)
 3:
 4: (* PASS5 MUST PRECEDE PASS4!    OTHERWISE LOSE POINTER TO S
 5: Unmatched lines in OA represent deletes; unmatched NA lines are
 6: inserts.  Ignoring these, the matched lines should increase
 7: monotonically.  When they don't, when a discontinuity appears,
 8: a block-move is present.  It is unclear how to record such moves
 9: in a difference file, so here we find them and "unmatch" the
10: moved block, converting it into a delete/insert.
11:
12: There is a figure/ground ambiguity -- any block move can be
13: seen as a move-up of some lines or a move-down of others.  We
14: select the smaller of the two groups to be the "moved" group. *)
15:
16: PROCEDURE PASS5;
17: VAR O,N: LINENUM;
18:
19: (* a discontinuity starts at OA[O] and NA[N].  figure out whether
20: fewer lines have been moved up to NA[N] or down from OA[O],
21: and convert the smaller group to inserts and deletes. *)
22:
23: PROCEDURE RESOLVE(VAR O,N: LINENUM);
24: VAR XO,XN,FIRST,LAST: LINENUM;
25:     T: INTEGER;
26:     S: SYMNUM;
27: BEGIN
28:
29: XO:=O; (*measure the block starting at OA[O] *)
30: REPEAT
31:   T:=OA[XO].INDEX + 1;
32:   XO:=XO+1
33: UNTIL (T <> OA[XO].INDEX) OR NOT OA[XO].MATCHED;
34:
35: XN:=N; (*measure the block moved up to NA[N]  *)
36: REPEAT
```

logic in this loop so only a single scan of the symbol table is needed.

If memory is at a premium, the size of the symbol table can be reduced to about the same size as the input files! Contrary to intuition, most compares contain only about as many unique lines as the size of one input file. A protective check on the amount of heap and symbol table space should be added to the procedure "store." I have found that with a TPA of about 55K RAM, the program will compare two 700 line Pascal files.

I think that the author's application of this intriguing algorithm is inappropriate for microcomputers. An ed.com script to change an old file to a new file is not very useful outside the world of mainframe source code updates. Of greater interest is a general utility that displays line differences between two versions of a program, and DIFF can easily be altered to do so.

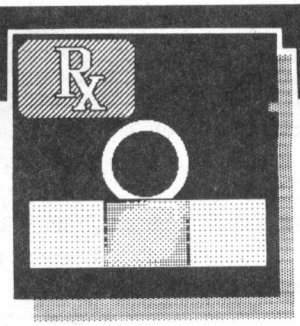
I do not at all mean to be critical of D. E. Cortesi. Quite the reverse, I found his article and program fascinating, and my comments reflect the fact that I have spent my time accordingly. Again, thank you.

Steven I. Rothman
President
Copper Creek Systems
P.O. Box 275
One Copper Creek Road
Glenwood, NM 88039

```
37:    T:=NA[XN].INDEX + 1;
38:    XN:=XN+1
39:  UNTIL (T <> NA[XN].INDEX) OR NOT NA[XN].MATCHED;
40:
41:  (* Unmatch a block of matched lines.  We need the symbol table index
42:  here, and find it by scanning the table.  This expensive operation
43:  could be eliminated by retaining the symbol index in OA
44:  ABSOLUTELY MUST PRECEDE PASS4, (WHICH CLOBBERS POITER INTO S)
45:  CHANGED FOR AN INEXPENSIVE SINGLE PASS *)
46:
47:  IF XO-O < XN-N THEN
48:    BEGIN  (*move block down*)
49:      FIRST:=O;    (*first oline to convert*)
50:      LAST:=XO-1;  (*last oline to convert *)
51:      O:=XO;       (*for restart of scan   *)
52:    END(*THEN*)
53:  ELSE
54:    BEGIN  (*move block up*)
55:      FIRST:=NA[N].INDEX;  (*first oline *)
56:      LAST:=FIRST+XN-N-1;  (*last oline  *)
57:      N:=XN;               (*restart scan*)
58:    END;(*ELSE*)
59:
60:  S:=0;
61:  FOR T:=FIRST TO LAST DO BEGIN
62:    WHILE (ST[S].OLINE < FIRST) OR (ST[S].OLINE > LAST) DO S:=S+1;
63:    XO:=ST[S].OLINE;
64:    XN:=OA[XO].INDEX;
65:    WITH OA[XO] DO BEGIN MATCHED:=FALSE; INDEX:=S; END;
66:    WITH NA[XN] DO BEGIN MATCHED:=FALSE; INDEX:=S; END;
67:    S:=S+1;
68:  END;(*FOR*)
69:  END;(*RESOLVE*)
70:
71:  BEGIN(***PASS5***)
72:  O:=1;  N:=1;
73:  REPEAT
74:    WHILE NOT OA[O].MATCHED DO O:=O+1; (* skip deletes *)
75:    WHILE NOT NA[N].MATCHED DO N:=N+1; (* skip inserts *)
76:    IF (N > NEWMAX) OR (O > OLDMAX) THEN EXIT;
77:    IF OA[O].INDEX=N THEN BEGIN O:=O+1; N:=N+1; END
78:    ELSE RESOLVE(O,N); (* discontinuity *)
79:  UNTIL FALSE;
80:  END;(*PASS5*)
81:
```

End Listing

DR. DOBB'S CLINIC



by D.E. Cortesi, Resident Intern

The Sky Is Falling

Well, maybe not the sky, but most of the world's information processing systems will be falling soon. Or so say Jerome and Marylin Murray, authors of *Computers in Crisis* (Petrocelli Books, 1984, $32.95), a copy of which came to this office for review. The Murrays' thesis, as summarized in sensational terms on the book jacket and in its first chapter, is the answer to a computer-illiterate journalist's prayer—an impending disaster of major proportions caused by blind reliance on computers, with overtones of government inertia and commercial callousness. It taps into so many modern archetypes that we predict that the Murrays will soon be appearing on talk shows everywhere.

Since considerably more hot air than illumination will be generated in the ensuing flap, we thought our readers would like a technically literate review of the book.

A FIPS standard determines how data should be stored in computer files. It applies primarily to programs written in COBOL or RPG that run on IBM 370 mainframes. In such programs, decimal numbers usually are stored in what IBM calls packed decimal format: BCD digits with the least-significant four bits reserved for a sign digit. The FIPS standard says dates should occupy four bytes in the form *Oyymmdds*, where *yy* is the year, *mm* the month, *dd* the day, and *s* the sign. Since these dates are also decimal numbers, they may be compared with each other.

An awful lot of such dates appear in data bases everywhere. The Murrays say that on Monday, January 3, 2000, a lot of programs are going to start doing incorrect things because a date of 0000103 is numerically less, not greater, than one of 0991231. And if nothing is done, they will probably be proved right. We find it hard to believe that nothing will be done, although this book may serve a useful purpose in focusing attention on the problem sooner than might otherwise happen.

The Murrays' main point is how to fix the problem. Once past the sensational claptrap of the first chapter, the reader finds a sound, interesting tutorial on the history of the calendar and a sensible proposal for a new standard format for dates. The suggested form is *yyyyddds*. This "neo-Julian" date, as packed decimal, fits in the same four bytes as the old version, so you won't have to reconstruct your 20-gigabyte IMS data base.

There you have the meat of the book; we've just saved you $32.95. The bulk (and we use the word advisedly) of the volume is a set of subroutines for the conversion and manipulation of dates. These are presented in what the authors call pseudo-code. The usual purpose of pseudo-code is to clarify and stress the structure of an algorithm. Unfortunately, the Murrays use a pseudo-code style that is actually *less* structured than typical BASIC (they deny themselves even the use of a for statement).

The subroutines are also presented in 370 assembly language. Fortunately for the publishers, most reviewers won't be able to read IBM ALC. We do know a TRT from a ZAP, and we weren't impressed. The routines aren't reentrant, they use numeric condition codes for branches, they take their arguments in labeled variables not registers, and they commit other coding sins.

The grand catastrophe forecast by this book applies only to programs that store dates in the FIPS format for packed decimal; those that use a different encoding for dates are exempt or at any rate will get no help here. Recommended only for those who work with software that might be affected—and then only if they can recover the ridiculous cover price from their employers.

APL: A Rigorous Approach

Now, if you want a book with real *body*, let us recommend the *Draft Proposed Standard: Programming Language APL*, available from the ACM Order Department, P.O. Box 64145, Baltimore, MD 21264 for $25. APL documentation has always been clear and precise, and that tradition has been carried forward even into such anti-linguistic surroundings as an ANSI subcommittee, ordinarily the milieu of the mumble.

That it is written in tight, grammatical English is not the most remarkable thing about this standard. What makes it really unique is that it specifies in full the semantic behavior of an APL implementation. Some standards restrict themselves to syntax; others attempt to specify behavior only for selected parts (as when the Pascal standard struggles to spell out the behavior of files). This standard claims to be "essentially an implementation of APL in a restricted form of English." When we first looked at it, we exclaimed, "Hey, it's a functional spec!" That impression lingers. If you wanted to write your own APL interpreter, you could almost do it by coding the lines of the *Draft Standard* into a programming language. It's that explicit.

Microsoft's MacBug

We asked if newer Microsoft BASICs would still misinterpret the command:

```
SAVE "PROG,A
```

Our esteemed editor tried it on his PC and it didn't fail—or at least it didn't write a binary file named PROG,A. But

Richard D. Norling of Washington, DC, tried it on a Mac. "The old bug still lives," he writes, "in the version Microsoft cooked up for the Macintosh. However, this particular bug does not cause further trouble [from the filename]. The Finder, Apple's disk operating system, is capable of handling filenames containing commas."

Norling had another problem, though. He says there is "a killer bug that causes only part of your program to be saved to disk if you touch a key while the save is in progress. No warning, and no way to recover since you aren't likely to discover the problem until you try to load the program at your next session. Microsoft actually issued a new version that fixes this bug in April [but] has left [users] to discover the bug on their own. You are told about the new version only if you call to complain."

We don't see the difficulty. If you keep your hands on the MacMouse where they're supposed to be, you won't be hitting the keyboard during a save anyway, right?

BASIC Haters Unite

Perry Dinsmore of Knoxville, TN, likes controversy and wants to whip some up in this column. We don't mind. All right, Perry, speak your piece.

"Discussions of new computer languages correctly point out the advantages of these languages over older ones such as BASIC. The question remains as to whether there is anything that can be done in a newer language that cannot be done, with a little more effort, in BASIC.

"I would like to challenge anyone to write a program in any language whose end results cannot be duplicated in BASIC. If such a program proves impossible, then a second challenge is to write the non-BASIC program that requires the longest BASIC program to emulate it. For example, if a 12-line program in another language required 120 lines (of comparable length) of BASIC to do the same thing, the expansion factor would be 10. What is the largest possible factor and which other language achieves it?"

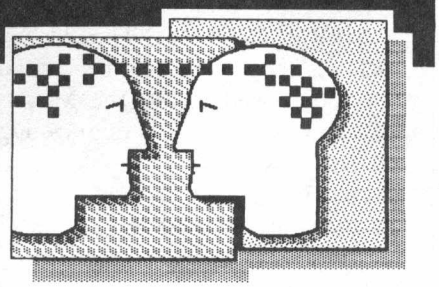
If you want to respond to Perry's challenge, please don't just send a sample of code in Forth, APL, C, or whatever, and say "match that." Life is too short for us, and probably for Perry, to sit around coding up BASIC solutions to other people's programs. You gotta include *both* programs, the short one and the BASIC version you think is equivalent.

Throughputting

We've had 10 or so letters from people who've measured the PIP or COPY throughput of their disk systems. We plan to recap all the numbers in the January Clinic. When you read this, we'll have already written that column, but there's no reason we wouldn't return to the subject again if more numbers came in. So far we haven't got anything for a Mac nor for any of the appliance machines like Atari or Commodore. Nor for an Apple under any DOS. Comparative throughput figures for a native Apple DOS versus a CP/M softcard would be interesting. We have one extremely impressive figure from a Zenith Z-100 running ZDOS (MSDOS 1.1); it would be nice to have corroboration of it.

DDJ

CP/M EXCHANGE



by Robert Blum

I know of nothing more wasteful than setting aside an idea because the difficulty of transforming it into a program seems to outweigh its benefit. The proliferation of data base packages over the past several years proves that I'm not the first and certainly won't be the last to feel this way.

I've tried almost every data base package available for CP/M-80; I prefer to call them application generators. Each has its own unique merits, and all are capable not only of making the transformation from idea to program less time-consuming than a solution based in BASIC or some other high-level programming language, but also of making the job easier to program. For me, however, the benefit of using one of these packages is not great enough to make me switch from assembly language for almost all of my work.

Perhaps I would be singing a different tune if I had not found the right programming tools. Fortunately for me I did, and I couldn't be happier, writing programs encumbered by only my own shortcomings.

Programmer's Tool Chest Improved

SYSLIB, the most complete collection of assembly language subroutines that I am aware of, is now available in its third revision—SYSLIB3. In this release all known bugs in the public domain revision, level 2, have been corrected—there were only a few—and a number of new subroutines have been added. The library has also been restructured to operate completely stand-alone, without any external assistance from other libraries.

Achieving this goal has meant that a few of the subroutines from previous versions are no longer available. They were intended for use in conjunction with ZCPR, a Unix-like shell written also by Richard Conn, and required support modules from that product to operate properly. In the current release of SYSLIB, all interdependencies between it and ZCPR have been severed. The subroutines contained in SYSLIB are now fully supported within themselves, and any ZCPR-specific subroutines or support modules have been placed into a separate library that is distributed with the ZCPR product.

As I have come to expect, the many new features added to this release of SYSLIB are well thought out and address some deserving areas of the CP/M interface. One of the new features, Switched Output, makes it possible to dynamically switch character output to the console or printer, or simultaneously to both, simply by changing a single memory variable. For example, to display the HL register pair as five decimal characters on the printer requires only the following three instructions:

```
LD    A,80h          ;direct output to
                     ;printer
LD    (SCTLFL),A     ;put into memory
                     ;variable
CALL  SHLDC          ;output HL
                     ;register pair
```

To direct the output to the console or simultaneously to the printer and console is as simple as changing the output direction byte to the proper value and storing it into the memory variable SCTLFL. You can use this same dynamic switching logic for many of the other character output and data format conversion subroutines as well.

The disk I/O routines have been enhanced to include random sector access and buffered single-byte disk I/O, utilizing buffers defined within (and the size chosen by) the application program. You can realize dramatic speed improvements when data is buffered in memory blocks of at least the same proportion as the physical disk sector size. This is especially true when multiple files are open at the same time. Next month I will go into a more elaborate discussion of how buffered disk I/O works and why the throughput gains are so great, as evidenced by several benchmark tests that I have run.

Documentation for SYSLIB now is contained in over 20 on-line help files. Each of the files has been rewritten for this release to explain more completely how to use each of the over 300 subroutines. In prior releases of SYSLIB, the on-line help facility was augmented by a large volume of printed documentation. Although unavailable at this time for SYSLIB3, I understand that efforts are being made to provide printed documentation at a future date for those who desire it.

Release 3 distribution of SYSLIB and ZCPR is being handled through Echelon, Inc., a company headed by Frank Gaudé, the author of DISK7, COMM7, and other notable public domain programs. Included with the purchase of the four-disk set containing SYSLIB is a newsletter subscription dealing with the Echelon family of products. Even though distributed commercially, SYSLIB is priced at only $29.00, which includes complete source code for each of the subroutines and a license to use the library in any nonprofit way you may desire.

I've been using SYSLIB for over a year now with only favorable results to report. For the experienced programmer, I know of no better way to reduce the coding redundancy inherent to assembly language programming and interfacing to CP/M. For the beginner, I cannot think of a better way to learn. For further information on the products mentioned here, contact Echelon, Inc., 101 First Street, Los Altos, CA 94022.

CP/M V2.2: PIP Patch

Trying to save time by not verifying the output when copying files with PIP will eventually lead to files that cannot be read later on. Rushing seems to be my normal pace, and it began to take its toll in the number of files I was recreating because I was more intent on getting done than spending a few extra minutes to ensure that my files were secure—that is, until I put into PIP the following patch, which forces the verify (V) option on at all times.

This patch is not approved by DRI; therefore, I suggest that you exercise extreme caution when installing it on your system. The installation steps are straightforward and shouldn't take more than a few minutes, but do be mindful not to overwrite anything of value.

The listing (at right) shows the steps I took to install the verify patch on my system. The first step is to verify that the PIP program being modified is version 1.5; this is the only version of PIP that I know the patch to work with successfully. To verify your version number, first load PIP.COM into memory with SID or DDT and display memory between 200H and 240H. At 0233H should begin the version number, which must be 1.5. Next change the single byte at 0B34H from 1FH to 37H. Finally save the modified program back onto disk under a new name ready for testing.

To ensure that I had entered my patches correctly, I timed how long it took to copy the same test file with an unaltered version of PIP using the verify option and again with the newly altered version of PIP. The two resulting times were practically equal, which satisfied me that my modifications were correct.

DDJ

Listing—PIP Verify Patch

```
B>SID PIP.COM
CP/M 3 SID — Version 3.0
NEXT MSZE PC END
1E00 1E00 0100 C8FF
#D200,240
0200 20 20 20 43 4F 50 59 52 49 47 48 54 20 28 43 29 COPYRIGHT (C)
0210 20 31 39 37 39 2C 20 44 49 47 49 54 41 4C 20 52 1979, DIGITAL R
0220 45 53 45 41 52 43 48 2C 20 20 50 49 50 20 56 45 ESEARCH, PIP VE
0230 52 53 20 31 2E 35 03 01 06 01 00 24 24 24 20 20 RS 1.5.....$$$
0240 20
#SB34
0B34 1F 37
0B35 D2 .
#WPIPNEW.COM,100,1E00
003Ah record(s) written.
#G0
```

End Listing

Adding Primitive I/O Functions to muLISP

by Michael Carter

One advantage of writing your own software is that you can tailor it to your personal taste. However, writing software is a time-consuming task, and most of us depend upon proprietary packages for most of our software needs. Although the range and quality of software are improving constantly, a proprietary package seldom performs exactly as we want. Usually we conform to the constraints of the package, but occasionally the gap between what we want and what we get is so large that it is worthwhile modifying the package.

This article describes my attempt to overcome a fundamental gap in a proprietary package. Although I deal with a particular example of a particular language, the general principles I discuss will apply to any modification. These principles concern where to locate the modifications, how to pass information to them, and how to recover information from them.

The language that is the subject of this article is LISP. I shall refrain from making a case for the use of this language. Suffice it to say that LISP is one of the earliest computer languages and has remained the mainstay of artificial intelligence. LISP or its derivatives will become more common on microcomputers as the applications become more sophisticated.

Michael Carter, Centre for Economic Policy Research, The Australian National University, P.O. Box 4, Canberra, ACT 2600, Australia.

"The principles I discuss for modifying proprietary software concern where to locate the modifications, how to pass information to them, and how to recover information from them."

The Problems

The muLISP implementation of LISP, developed by the Soft Warehouse and distributed by Microsoft, offers more than just a LISP interpreter. It is a LISP program development system complete with its own screen editor. This well conceived and executed package is a joy to use once you get the hang of it. Missing from the package, though, are any facilities for primitive input and output. muLISP has the usual console input and output functions, functions for printer output, and buffered disk I/O, but it provides no way to directly control any of the other I/O ports typically found on a microcomputer. You would need such a facility, for example, to control a robot—a task for which LISP is admirably suited. All that is required is two additional functions: IN[port] and OUT[port,byte].

Fortunately, the authors of muLISP foresaw the need for additional functions and provided a means of linking them into muLISP. They left room for four jump instructions starting at location 103H. Up to three arguments can be passed to each function in the HL, DE, and BC register pairs. The value of the function is returned via the HL register pair. (Since muLISP uses address typing to determine a function's type, all machine language routines must begin in low memory; hence the jump table at 103H must be used.)

In attempting to use these facilities, the programmer faces three problems:

(1) Where in memory to locate the machine language functions

(2) How to interpret the arguments that are passed
(3) How to return a value to muLISP

I will elaborate successively on each of these problems and my own solutions. Basically the task is to add two new functions to muLISP (IN and OUT) in such a way that the programmer can regard them as first-class LISP functions.

My discussion of the first problem and its solution is rather elaborate since it has general application and should be of interest to many people who are still becoming familiar with CP/M. However, neither problem (2) nor (3) is specific to LISP or muLISP. Modifying or expanding the facilities of any proprietary program will require some investigation of the means by which information is stored and the suitable adaptation of the package's facilities to new uses. While the particular solution presented here is specific to muLISP, I hope that readers will be able to draw some general principles from my experience that will help them in modifying other packages. Therefore, this article may be considered a general primer on interfacing new functions to existing programs. I used Z80 mnemonics throughout from personal preference, but the only specifically Z80 instruction is the block move LDIR, which you can readily rewrite in 8080 code.

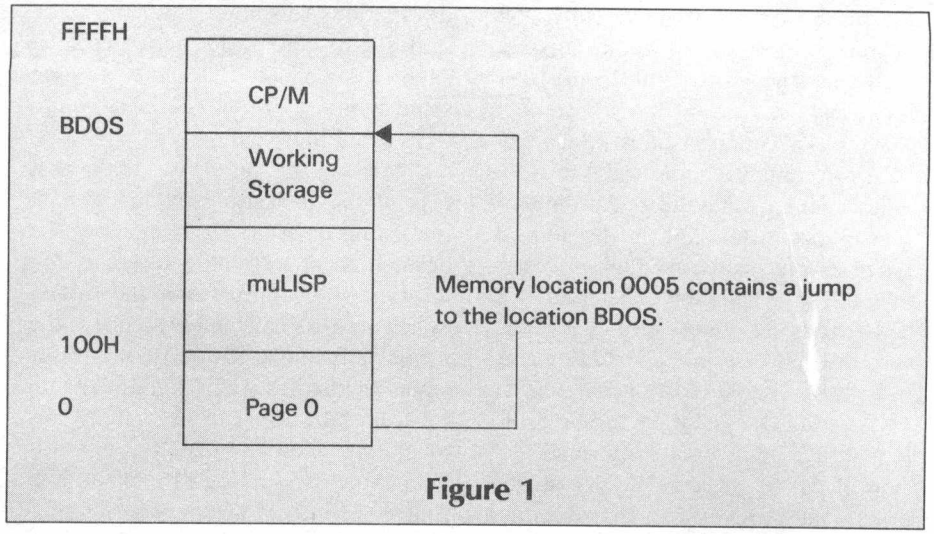


Figure 1

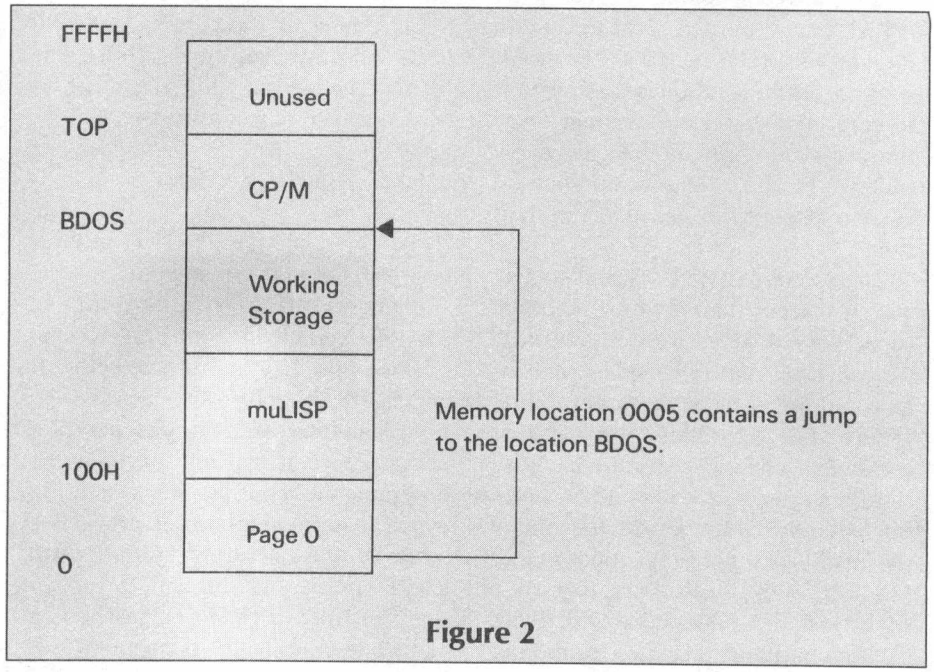


Figure 2

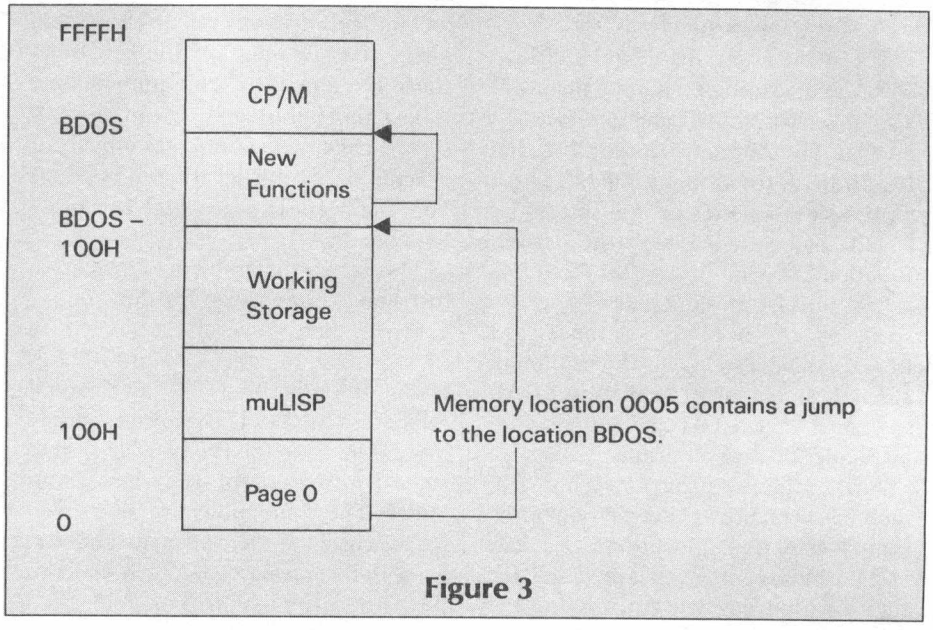


Figure 3

Locating the Functions

The first problem is quite general and one that confronts any programmer attempting to add to the functions of a CP/M program: Where can we find room in the computer for the additional code? The memory map for muLISP running in a standard CP/M system is illustrated in Figure 1 (at right). muLISP uses all the memory between 100H and the BDOS for its own purposes; muLISP itself occupies 10K starting at 100H; and muLISP uses the remainder of the available memory for data storage. This memory usage is typical of CP/M programs.

In some systems CP/M is configured to operate in less than the total available memory, leaving a block of RAM at the top of the memory space (Figure 2, at right). You may use this RAM to hold special-purpose software such as video display or printer drivers, as a printer buffer, or not at all. Whatever

the reason for its existence, any such memory above CP/M could serve to contain the additional muLISP functions.

For obvious reasons most CP/M systems are configured to use all the available memory, and the operating system is located as high in the address space as physically available memory will allow. Remember also that you can readily adjust the size of a CP/M system (line TOP in Figure 2) by running the provided CP/M utility MOVCPM. One solution to our problem would be to create a special version of CP/M for muLISP—a version of the operating system that does not occupy all the available memory, leaving a block of unused RAM to contain the new muLISP functions. The disadvantage of this solution is that we have to cold boot (reset) the system every time we want to run muLISP and every time we finish so that the appropriate version of CP/M is loaded at the right time.

This is not only tedious, it is unnecessary. Rather than creating a special version of CP/M for use with muLISP, we can have muLISP modify the system appropriately whenever it is run. Figure 3 (page 19) illustrates this more convenient and elegant solution. With a little subterfuge we create a "hole" immediately below CP/M to contain the machine language functions so that muLISP thinks it is operating in a slightly smaller system than it really is. Consequently it takes up somewhat less of the available memory, leaving space for the new functions.

All CP/M programs should interface with the operating system by means of a single entry point at memory location 5; that is, programs can utilize the functions provided by CP/M (e.g., write to CONSOLE, open disk file) by a call to location 5 with the desired function encoded in register C. Location 5 in its turn contains a jump instruction to the base of the CP/M BDOS, which also marks the beginning of CP/M in memory (see Figure 1). All space between 100H and this point is available for user programs.

Furthermore, programs (muLISP included) can use the destination of the jump instruction at location 5 to calculate how much memory is available for their own purposes. By modifying the destination at location 5, we can force muLISP to confine its activities to a smaller region of RAM, thus freeing some memory for the functions IN and OUT (see Figure 3).

Achieving this subterfuge is the first task of the subroutine INIT shown in the listing on page 23. It modifies the destination at location 5 to point to a location 100H lower than the BDOS. When muLISP calculates the size of the available memory, it will use this lower value. In this way we trick muLISP into regarding the 100H bytes of RAM immediately below the BDOS as unavailable, which frees 100H bytes for I/O routines. But how do we persuade muLISP to execute this piece of code?

Like all CP/M programs, muLISP loads and starts execution at 100H. The first instruction is a jump to another location, which we will call START:

```
0100    jmp     START
```

By patching in a jump to INIT at location 100H of muLISP and finishing the subroutine INIT with a jump to START, we convince muLISP to execute the initialization routine before it begins executing its own code.

The address at location 5 serves a dual purpose: it indicates to the user's program the size of the system in which it is running, and it provides a pathway for operating system calls. Therefore, we must provide a further jump to BDOS as the first instruction of this reserved page of memory (see listing). In this way a call to location 5 will find its way correctly to the BDOS, albeit after two jumps. The new functions IN and OUT can immediately follow the jump to BDOS in this reserved page of memory assembled for execution in this location. Locations 103H and 106H are patched to jump to IN and OUT.

This solves the problem of a location for the new functions. But how do we get them there? The simplest way is to tack them on to the end of muLISP following the function INIT and to add a relocator to INIT. The relocator simply moves the code for IN and OUT up to its ultimate destination at BDOS: 100H. These functions are assembled for execution at their ultimate address. This is the purpose of the pseudo-op .PHASE in the listing. This relocation facility, which is available in the MACRO-80 assembler package, makes this procedure easy. It is not essential, however. The functions can be assembled with ASM at their execution address, loaded with DDT, and then moved down to the end of muLISP for saving on disk.

The patch in the listing is assembled for a particular system size, implied by the address of BDOS. If the available memory changes, the patch must be reassembled to fit the altered system. I chose this implementation for simplicity, since I do not frequently alter my system size. You can overcome this restriction but at the cost of greater complexity. Interested readers might like to consult the articles by Gary Kildall (*DDJ*, February 1978) and John Palmer (*DDJ*, December 1981) on automatic relocation. In this particular example you could easily rewrite IN and OUT in relocatable form since the only absolute reference is the call to GETNUM.

My discussion so far has applied quite generally to CP/M. Now, however, it is necessary to delve more deeply into the structure of muLISP.

Evaluating the Arguments

I said earlier that up to three arguments can be passed to a machine language function in the HL, DE, and BC register pairs. It is important to note that the *value* of the argument is not passed to the machine language function but rather a pointer to its internal representation. To interpret the arguments, you must understand how data is represented internally in muLISP. The following information is taken from Section II of the *muLISP Reference Manual*.

muLISP recognizes three primitive data types: *names, numbers*, and *nodes*. Each has a different internal representation. To simplify the machine language functions, I have restricted them to accept and return numbers only. Note that this does not restrict their generality, since any data type can be transmitted as a sequence of numbers. I will confine my attention here to the numbers data type and merely note that the other data types have a similar representation. Those interested in further details should see Section II of the muLISP manual.

A number (in muLISP) consists of

three consecutive pointers:

number	sign	vector

A pointer is a 16-bit value (2 bytes) that contains a memory address—a pointer to another location. The first pointer (number) points to itself; that is, it contains its own address. The second pointer or cell (sign) indicates whether the number is positive or negative. The third cell (vector) points to the machine representation of the numerical value of the number.

In contrast to most languages, integers in muLISP are not confined to 16- or 32-bit values. In fact, numbers up to approximately 10^{611} can be represented exactly by allowing the length of the binary representation of a number to be arbitrary and including the length in that representation. The machine representation of the numerical value of a number consists of a single-byte byte counter preceded by the number of bytes required to express the number in binary (including the byte counter). The vector cell is a pointer to this in binary representation—in fact it points to the byte counter. Contrary to the manual, the byte counter follows rather than precedes the bit vector, which itself is stored most significant bit first. The number zero is a special case; it is represented by a single byte, the byte counter, which has the value one.

The argument that is passed to a machine language function in one of the register pairs (assuming a numerical argument) is a pointer to the number cell of the internal representation of the number. For example, suppose that the first argument is the value six. Then the register pair (HL) will point to a muLISP number that has the value six. The internal representation of this is depicted in Figure 4 (page 22).

We are now in a position to understand the subroutine GETNUM in the listing. This subroutine returns the numerical value of the argument that is pointed to by the HL register pair. GETNUM skips over the number and sign cells to the vector cell. The value of this pointer is loaded into the HL register pair, which now points to the byte counter. If the value of the byte counter is one, the value of the number is zero, which GETNUM returns. If not, GETNUM returns the value of the immediately preceding byte.

In this particular application all numbers (port numbers and byte values) lie between 0 and 255. You may substantially simplify GETNUM by taking advantage of this knowledge. Thus, it is implicitly assumed that the byte counter will have a value of one or two. If not, GETNUM returns the value of the number mod 256.

Returning a Value to muLISP

The value of a machine language function is returned via the HL register, but the HL register does not contain the value itself. Rather it contains a pointer to the appropriate data structure. In the case of the input function, this structure will be the address of a bona fide muLISP number; that is, the input function must return a pointer to a sequence of three cells (number, sign, and vector),

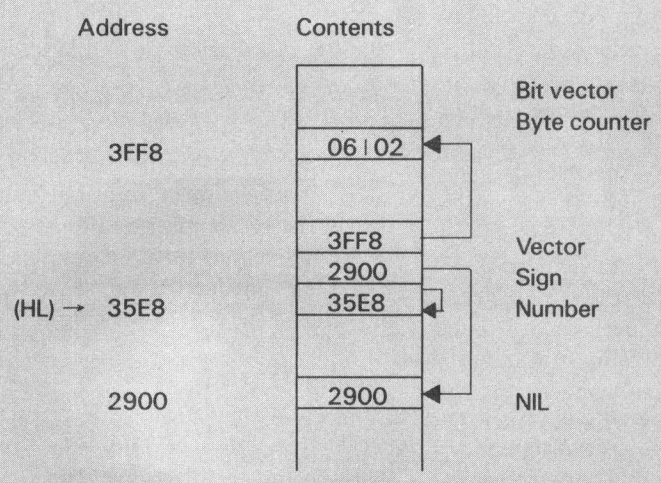


Figure 4

Memory contents are shown as words (16 bits) with the order of the most and least significant bytes reversed, except in the case of the byte counter and the bit vector, which are shown as bytes. The bit vector precedes (has a lower address than) the byte counter. Location 2900H is a special atom: NIL. The sign of a nonnegative number always points to NIL.

```
IN:         call    GETNUM
            ld      c,a
            in      a,(c)
            ld      hl,NUMBER     ;point to number to be returned
            ld      (VALUE),a     ;bit vector
            or      a             ;set flags (i.e., is VALUE = 0)
            ld      (COUNT),1     ;byte counter if VALUE = 0
            ret     z
            ld      (COUNT),2     ;byte counter if VALUE > 0
            ret
NUMBER:     dw      NUMBER
SIGN:       dw      NIL
VECTOR:     dw      COUNT
VALUE:      db      0
COUNT:      db      1
```

Figure 5

the last of which contains the address of the binary representation of the desired value. Achieving this turned out to be a nontrivial task, and it is worth exploring briefly two false paths I followed for the insight they provide into the workings of the interpreter.

My first attempt was to build up the appropriate data structure as part of the code of the I/O functions themselves. IN had the form shown in Figure 5 (below). This attempt ran afoul of address typing. The certain region of address space allocated for names and numbers does not include the location of the machine language functions. Attempting to refer to a number outside this atom space had interesting but not highly desirable consequences.

The second possibility that I considered was to use one of the parameters to return a value. A valid muLISP number is passed as the port address. Once its value has been used, it is no longer required. Why not simply amend its vector value to reflect the value of the input port and return the same pointer? Why not indeed?

This suggestion is precluded by the constraint on muLISP machine language functions to be *call by value* functions, which means that the function arguments are evaluated before they are passed to the function. What is passed is a pointer to the value of that argument, a muLISP number that may be shared by many other objects in the LISP environment. Changing the value of this number changes all these values simultaneously. Again the results are interesting but not always desirable.

The final and ultimately successful tack was to discover how muLISP handled the same problem. Some tedious disassembly revealed the existence and location of a subroutine that converts the number in register A into a valid muLISP number. So IN simply jumps to this subroutine with the value it obtains from the port. In the listing this subroutine is labeled RETURN_A.

This method of returning a value to muLISP is simple. However, it has the disadvantage of requiring prior knowledge of the absolute location of the subroutine RETURN, which may differ from version to version of the interpreter. The next section provides some hints on finding the location of RETURN_A.

A final point: Every machine language function must return a value; that is, the HL register pair must point to a bona fide muLISP data object (e.g., NIL). The function OUT simply returns the value of the byte it has output.

Finding RETURN_A

The location of the subroutine RETURN_A can be found by disassembling the function MEMORY, the location of which can be found with the following muLISP command:

```
(GETD MEMORY)
```

Remember that the returned address will be in decimal unless you change the radix. In my system the code for MEMORY is as follows (with the radix changed to 16):

```
12D5   CALL  0650
       JP    NC,07BE
       LD    A,(HL)
       PUSH  AF
       EX    DE,HL
       CALL  0672
       JP    NC,12E5
       LD    (DE),A
       POP   AF
       JP    0FC6
```

This last address, 0FC6, is the location of the subroutine RETURN.

Conclusion

Adding some simple I/O functions to muLISP turned out to be more complex than I first envisaged. In the process I learned some interesting details about the internal structure of the interpreter. I pass on the results of my investigations in the hope that they may be of use to others who want to add other machine language functions to muLISP. The listing gives the requirements for the primitive I/O functions IN and OUT.

My efforts may also be of value to other programmers. The section that deals with obtaining a suitable block of memory for the machine language functions applies quite generally to CP/M programs. Although the remaining sections are more specific to muLISP, the general concepts addressed are worthy of attention by a wider audience. Any LISP implementation is likely to follow a similar structure, while any high-level language is likely to provide similar problems.

DDJ

muLisp Listing (Text begins on page 18)

```
                        ;Modifications to muLISP to provide primitive I/O  functions
                        ;The function and operation of this code is fully explained in the accompanying text
                              .Z80
0000'                         aseg
0005                    GOBDOS      equ     5           ;CP/M entry point
0134                    START       equu    0134H       ;entry point to muLISP
0FC6                    RETURN_A    equ     0FC6H       ;muLISP subroutine
2880                    ENDLISP     equ     2880H       ;end of original code
A300                    BDOS        equ     0A300H      ;depends on memory size

                              org     100H
0100    C3 2880               jp      INIT              ;muLISP jump table
0103    C3 A203               jp      IN
0106    C3 A20C               jp      OUT
```

```
0109    C3 0109          USER3:  jp      USER3
010C    C3 010C          USER4:  jp      USER4
                                 org     ENDLISP
2880                     INIT:
2880    21 A200                  ld      hl,BDOS - 100H      ;patch GoBDOS with lower address
2883    22 0006                  ld      (GoBDOS+1),hl       ;see Figure 3
2886    EB                       ex      de,hl               ;and relocate functions
2887    21 2892                  ld      hl,CODE             ;ie. move machine language functions
288A    01 0027                  ld      bc,LENGTH           ;up to memory page immediately
288D    ED B0                    ldir                        ;below BDOS
288F    C3 0134                  jp      START               ;of muLISP

2892                     CODE    equ     $
                                 .phase  BDOS - 100H         ;this is the code of the machine language
                                                             ;functions assembled to reside at BDOS - 100H
A200    C3 A300                  jp      BDOS                ;jump for BDOS calls

A203    CD A219          IN:     call    GETNUM              ;get port number
A206    4F                       ld      c,a
A207    ED 78                    in      a,(c)               ;input value at port
A209    C3 0FC6                  jp      RETURN_A            ; return to muLISP

A20C    D5               OUT:    push    de                  ;save output value for return value
A20D    CD A219                  call    GETNUM              ;get port number (pointed to by hl)
A210    4F                       ld      c,a
A211    EB                       ex      de,hl
A212    CD A219                  call    GETNUM              ;get value to output
A215    ED 79                    out     (c),a               ;output value to port
A217    E1                       pop     hl                  ;return value (from stack)
A218    C9                       ret
A219                     GETNUM:
                         ; returns a muLISP number mod 256
                         ; on entry, hl points to number
A219    23                       inc     hl                  ;skip over first two (16 bit) fields
A21A    23                       inc     hl                  ;see Figure 4
A21B    23                       inc     hl
A21C    23                       inc     hl
                                                             ;hl now points to pointer to the bit vector
A21D    7E                       ld      a,(hl)              ;load pointer in hl
A21E    23                       inc     hl
A21F    66                       ld      h,(hl)
A220    6F                       ld      l,a
                                                             ;hl now points to the byte count
A221    7E                       ld      a,(hl)              ;load byte counter into a
A222    3D                       dec     a
A223    C8                       ret     z                   ;return 0 if byte count = 1
                                                             ;otherwise
A224    2B                       dec     hl
;point to LSB
A225    7E                       ld      a,(hl)
A226    C9                       ret
;return LSB
                                 .dephase

0027                     LENGTH  equ     $ - CODE
                                 end
```

End Listing

Program Monitor Package

Using Interrupts to Instrument Applications

by Alan Bomberger

The optimization of programs for minimum execution time is a passion for some programmers and a necessity for others. I sometimes argue that the three phases of a good programmer's development are: first, optimize for minimum number of instructions; second, optimize for minimum execution time; and last, optimize for readability. Once programmers reach the last phase, they can begin to use computer aids to optimize for execution time and program size. The package described here is one of those aids.

Many quickly written programs find their way into production products; when they do, they often need to be rewritten. Sometimes small, nagging bugs are removed, sometimes features are added, but most often the program that worked well on a little data takes far too much time when given a full measure of data. Unguided attempts to optimize programs for speed often result in pointless modifications that only reduce readability of the program.

I recently examined a program of mine (a spelling checker) with an eye to optimizing its speed. I examined the code in places where I thought the program could be "tightened up" but felt that changes in those areas would produce buggy code with no obvious benefit. I needed to develop an instrumentation package to determine where the program spent most of its time.

The ideas behind such a package are not new, and the techniques for implementing the package on large systems are well known. I simply applied these ideas and techniques to a CP/M system and developed a Program Monitoring Package (PMP) for CP/M.

Overview

The principal technique for monitoring programs is to examine the program counter (the address of the fetch of the next instruction) while the program is running. When machines had front panels, a rather crude analysis consisted of watching the lights: the lights that glowed the brightest represented the address in memory from which most of the instructions were fetched. This technique, however, could locate only extremely time-consuming sections of code.

"Unguided attempts to optimize programs for speed often result in pointless modifications. I needed to develop an instrumentation package to determine where the program spent most of its time."

A more precise technique is to record the value of the program counter at random instants while the program is running. To record every value of the program counter is usually impractical, so the program is divided into small sections based on the address. A counter is incremented each time the program counter is seen within a section. Coarse samples might divide the program into 1024-byte sections, while fine samples might use sections as small as 32 bytes. Finer sampling implies more counters. Two thousand counters are needed to sample a 64K program with 32-byte sections.

Alan Bomberger, Poor Person Software, 3721 Starr King Circle, Palo Alto, CA 94306.

With enough samples it is possible to identify small sequences of time-consuming instructions. I once analyzed a heavily used program for hours to locate the code that was slowing it down. Coarse sampling isolated a subroutine that was consuming much more than its fair share of computer cycles. Detailed sampling within the subroutine finally isolated a divide instruction as the culprit. A rewrite of the subroutine without the divide instruction resulted in a 10% increase in program speed. In another case, coarse sampling isolated a sort subroutine with a very poor algorithm. A rewrite of the sort made the program run 10 times faster.

Sampling may reveal that no isolated areas of the program are particularly inefficient. In these cases an examination of the algorithms involved is necessary to determine if there are more efficient ways of solving the problem.

Later in the article I will discuss the pitfalls of sampling (random and otherwise) the program counter. It is, however, a good technique for isolating the immediate areas in a program that must be examined in detail. Other techniques also analyze the behavior of a running program. One popular technique is to use a special compiler that inserts subroutine calls to a recording feature at significant places in the program. The inserted subroutine records the events for later analysis. The analysis shows DO LOOP counts, SUBROUTINE counts, and even counts on each branch of an IF statement.

The major advantage of this technique is that it enables a complete flow analysis, with exact identification (in terms of the source language) of time-consuming code. The major disadvantage is that the technique depends on the availability of a special compiler for the language being used. By contrast, the interrupt-driven sampling technique is applicable to any program in any language. Its major disadvantage is that it usually provides no knowledge of the program structure, requiring the programmer to correlate the program counter values with particular sections of code.

Interrupts

Just how does this technique sample the program counter? Most computers, large and small, have a feature that allows some sort of timer to interrupt the program. As its name implies, an interrupt is an unpredictable change in the status quo. In other words, some program segment in the computer memory is executed out of sequence and without the knowledge or cooperation of the main program.

Timer-based interrupts are the basis for program monitoring. The interrupt executes a program segment that examines the program counter of the interrupted instruction and increments a counter based on the section of the program executing at the time of the interrupt. The segment then restores the status of the machine to its exact state at the time of the interrupt and returns control to the program. The only effect on the program is that it runs a little slower.

Taking Control from BDOS

Because the instructions that the computer executes when an interrupt occurs are neither part of the application program nor CP/M, the programmer must find some technique to reserve a portion of the computer's memory for this purpose. Some systems have memory available at addresses beyond either the program or CP/M. Such systems, however, are becoming rare, and the programmer is likely to need a method that uses a portion of the memory normally used for programs. Such a method has been part of the CP/M lore for many years.

The instruction at location 5 of a CP/M system is a jump to the CP/M BDOS. Location 6 is the address of the lowest memory used by the BDOS; application programs can use any memory up to this point. When an application is not in control, the CCP (console command processor) is loaded immediately below the BDOS. To reserve a portion of memory for the interrupt program, the programmer must change the address at location 6. Once this is done, applications will not use the memory reserved for the interrupt program and the CCP remains in memory.

It is easy to see that the instruction at location 5 must now jump to the lowest address used by the interrupt program; this must be a jump to the address that was in location 6 before the change was made. This "bounce pass" does not interfere with the operation of CP/M.

Three additional details must be worked out. First, a program must load the interrupt program into the memory reserved for it. An initialization program usually accomplishes this, running as a standard application that moves a block of code from low memory to just below the CCP. This code must be assembled assuming a location below the CCP and not in low memory; assembly language constructs explained in Listing Two (page 32) accomplish this.

Second, an application program should be able to retrieve the samples of the program counter for analysis. The interrupt code adds two new BDOS function calls for his purpose. Because all BDOS calls now pass through the interrupt code on their way to the CP/M BDOS, it is possible to intercept certain calls and process them in the interrupt code itself. Each BDOS request is examined to see if it is one of the new functions. If not, the request passes on to the real BDOS. If it is a new function, the processing takes place in the interrupt code.

In this package one BDOS function zeros the counters and starts the program monitor. Another function returns a pointer to the counters and an indication of the address range of each counter. When the interrupt code interprets these BDOS functions, return is directly to the calling program without jumping to the real BDOS.

The third action is to modify the interrupt vector in low memory so that the PMP is entered each time a timer interrupt occurs. I will explain this action in detail further on.

Taking Control from BIOS

Several changes in the behavior of the BIOS are necessary. The WARM BOOT function must not overlay location 6 with the real BDOS address; it need not read in the CCP (which is protected from the applications by the value in location 6) or the BDOS. To intercept the BIOS WARM BOOT call, the programmer must change the address in the BIOS jump table.

Location 0 is a JMP to a vector of JMP instructions. The first of these is a jump to WARM BOOT. (Other jumps in this table jump to various I/O routines of the BIOS.) A jump to location

0 therefore causes a jump to the WARM BOOT routine. To intercept the WARM BOOT function, we need only change the first address in the BIOS jump table to point to our code.

To simulate the WARM BOOT, we use a BDOS function RESET DISK: the stack pointer is reset and location 6 is reset to the correct value (which points to our code). The reset of location 6 is necessary because some processors that use a similar technique of modifying location 6 do not reset it when done. DDT is an example.

Two other modifications to the BIOS jump table are necessary. Many systems do not expect interrupts to be enabled during CP/M operation. Interrupts do not bother CP/M, but they may bother the BIOS disk I/O functions: an interrupt during the transfer of data to and from the disk can cause a loss of data. To avoid this, many BIOS systems disable interrupts before performing the disk I/O. Not all of these systems, however, enable interrupts when they are done. Some systems do not bother with interrupts at all, under the assumption that they are never enabled.

For the program monitor to function without disturbing the operation of CP/M, it is necessary to assure that the system disables interrupts before entering the BIOS disk read and write routines and enables interrupts when the disk I/O is complete. We accomplish this by intercepting the jumps to the disk read and disk write functions of the BIOS. By modifying the jump table so that calls to the BIOS disk functions come first to the interrupt code, we can instruct the system to disable interrupts, call the real BIOS function, and then enable interrupts.

It is worth noting here that this technique uses additional stack space. What used to be a simple call to the BIOS becomes two calls: first the interrupt code is called and then the interrupt code calls the BIOS. This requires two extra stack locations. If an application does not have enough stack space, it may not run correctly. Eliminating the use of extra stack space is possible, but it introduces additional complications.

Borrowing Interrupts

The program monitor uses a timer interrupt to sample the program counter. Some CP/M systems already use a timer interrupt for some other purpose. For example, my system uses timers to simulate console interrupts. Others use timers to update clock values and the like. The program monitor works in these environments because it is aware that it may have to share the timer interrupt among several programs.

If a system uses timer interrupts, one of the fixed interrupt locations in low memory (0 – 38) is set up with the address of the timer interrupt routine. The occurrence of a timer interrupt calls this fixed location. The content of the fixed location usually is a jump to the program that handles the timer interrupt. When the interrupt program is complete, it enables interrupts and returns to the running program.

The program monitor uses the intercept technique to borrow the timer interrupt. The programmer changes the fixed location contents to a jump to the program monitor, which executes the code to record the program counter. When the program monitor finishes, it jumps to the old interrupt program instead of returning to the running program. The old interrupt program now runs as if it had been jumped to directly, returning to the running program when it is finished.

Again note that this interception may require additional stack space. I have coded the program monitor very carefully to use as little stack space as possible. Registers are saved outside of the stack and calls are minimized.

Before the program monitor can begin, it must determine whether the timer interrupt is in use or not. If not, the program must initialize the hardware that causes the interrupt and enable interrupts. If the timer is in use, the program only modifies the interrupt location to borrow the interrupt.

A test is necessary to determine whether the timer is running or not. This test borrows the timer interrupt and sets a flag if the timer goes off. If after a short wait the flag is on, the timer was previously running. Different techniques initialize the program monitor depending on whether the timer was running or not. All of these techniques are documented in Listing Three (page 44).

Sampling Theory without Mathematics

As the word implies, sampling is an incomplete recording of information. It is subject to a certain class of errors that relate to the relative frequency of the samples and the information that is sampled. Imagine a large program loop that takes 10 msec to complete. Sampling the program counter every 10 msec identifies only one portion of the loop. The complete identification of a loop requires knowing the bounds of the loop as well as its location. (The bounds of most loops can be identified from the program listing. However, determining if some portions of the loop take longer than others requires samples throughout the range of the loop.) If the same program loop were to take only 2 msec, samples every 10 msec might not identify it at all. Increasing the sample rate to every 1 msec would identify either loop correctly.

Most programs are a series of loops within loops and represent periodic processes. As seen above, periodic sampling of periodic processes can provide misleading information. A sufficiently high sampling rate will ensure that information is not lost but may require recording an excessive amount of information. Sampling at random intervals takes advantage of the benefits of a high sampling rate while limiting the amount of information recorded.

When sampling a 10 msec loop at 1 msec intervals, little information is lost if some of the samples are discarded. The sample rate of 1 msec ensures that loops as short as 2 msec are correctly identified, and discarding samples at random ensures that no information is lost over the long run.

Sampling at a fixed interval of 1 msec and discarding random samples is equivalent to sampling at random intervals that are a multiple of 1 msec. The minimum interval of the randomizing function determines the accuracy of the data based on periodic sampling. Random sampling or randomly discarding periodic samples does not increase the accuracy of the information but decreases the volume of data that must be recorded.

Random discarding of periodic samples is a method of simulating random sampling that can be implemented in software. This simulation is often nec-

essary because many small computers have timers with intervals fixed by wiring changes—these timers cannot be set at random intervals by software. The software simulation of random sampling is not without significant cost. It involves fielding a timer interrupt at the highest periodic rate and using a counter set at a random value to determine which samples are recorded. The software to save and restore status at the interrupt and to calculate the random value for the counter may involve as many as 100 instructions. A Z80 with a 4 MHz clock executes between 200 and 300 instructions per msec, so the effect of the sampling overhead under these conditions is a 30 – 50% decrease in speed.

Absolutely accurate data is unnecessary as long as the possible errors in the data are understood. With this understanding and with a desire to minimize the overhead involved in sampling, the PMP as implemented on my hardware uses periodic sampling with an interval of 3.3 msec (which is, conveniently, the shortest interval available with my NorthStar Horizon computer). This works out to about 300 samples per sec for a Z80 with a 4 MHz clock or about 1 sample every 1000 – 2000 instructions.

Because loops of less than 1000 instructions are common in programs coded in assembly language, the accuracy of sampling at this rate is not sufficient to rely totally on the output of the PMP when determining which areas of code to examine. The fact that samples represent a range of addresses rather than a specific instruction mitigates this problem. In a program divided into 64-byte ranges (1024 counters for a full 64K memory), 10 counters would represent a 300-instruction loop. The only way to establish the exact location of code inefficiencies, given this information, is to examine the code. The output of the PMP, however, is helpful in isolating areas that need detailed examination.

Analyzing Results

The first decision we must make when using the PMP is the size of the program section represented by each counter. Because this currently is determined at assembly time, we may need several versions of the PMP for the complete study of a program. The PMP allows a minimum program section size of 32 bytes, which requires 2048 counters occupying 4K of memory. If the program is small (thus requiring the 32-byte section size), there will be adequate memory for the counters.

The second decision is the selection of the sample rate. Most programs chosen for analysis with the purpose of increasing their efficiency will be long-running ones. The selection of the sample rate must achieve a balance between identifying small loops and keeping the counters from overflowing. In my case, the NorthStar hardware limits the sample rate choices.

Once we have configured the hardware and software for the PMP, the analysis of a program consists of four steps: The PMP must be loaded into the CP/M system, the program monitor started, the program run, and the monitor stopped and results printed. The system I use prints a matrix of counter values for each program section.

We then examine the program listing and identify each section of the program with a higher-than-average counter value. Be sure to note when only one portion of a loop has a high counter value. This usually indicates that the loop interval is close to the sampling interval, and we must be careful in drawing conclusions about the changes that must be made in that part of the program. It might help to add a long string of NOPs to such loops and rerun the program with the monitor. Adding the NOPs (or statements like A = A;) will change the length of the loop and increase the accuracy of the sample information.

Conclusions

I used the PMP to analyze the Poor Person's Spelling Checker after I had converted it (originally written for Z80) to 8080 and had made several enhancements (increasing the degree of compaction in the lexicon and adding hash chains to the word lists). See Listing One (below). I had expected the expansion and hashing routines to be a significant factor in the performance. What I discovered was that some sloppy code in a routine to follow chain links was responsible for most of the performance problem. I spotted other minor efficiency problems and reworked the code. The routine to follow chain links is still a major consumer of CPU resources, leaving me to conclude that further improvements will come only with redesign.

DDJ

Program Monitor Package (Text begins on page 26)

Listing One

An example of the PMP in use.

The Poor Person's Spelling Checker was monitored using 32-byte program sections while it was checking the spelling of this article. The sections of memory that were not used have been edited out of the listing to save space. The samples at high addresses are from the BDOS and BIOS.

```
A>pmp on
A>monitor -b
A>b:spell pmp.ddj pmp.mis

Poor Person's Speller (c) 1981,1983 Alan Bomberger
```

(Continued on next page)

Program Monitor Package (Listing Continued, text begins on page 26)

Listing One

```
0715   distinct words in text
Enter lexicon file name (.LEX  assumed) or <RETURN>
b:spell
Begin spelling check pass
Enter lexicon file name (.LEX  assumed) or <RETURN>
TM
BM
LM
RM
DS
CTR
S
READABILILITY
UNGUIDED
POINTLESS
INSTRUMENTATION
CP
M
MONITORING
PMP
OVERVIEW
LIGHTS
INSTANTS
K
SUBROUTINE
NON
PREDICTABLE
QUO
BDOS
BECOMMING
CCP
BIOS
JMP
O
DDT
MILLISECONDS
MILLISECOND
RANDOMIZING
SOFTWARE
INTERURRUPTS
SLOWDOWN
THUS
INDENTIFYING
NOP
A>monitor -e

        0  20  40  60  80  A0  C0  E0 100 120 140 160 180 1A0 1C0 1E0
  0     3                             14   D             2             6
 200  1D6 18B  36  57 25A 2D5  AD  1A   3       45  C2
 400   8F 324 224  A0 391 22C 35C 2AB  F2   4   6   2   3  C0 31E   9
 600       1
 800                                                     1
 A00
 C00
 E00
1000
1200
1400
1600
1800
1A00
```

```
1C00
1E00
          .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
          .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
          0  20  40  60  80  A0  C0  E0 100 120 140 160 180 1A0 1C0 1E0
C000
C200
C400
C600
C800
CA00
CC00
CE00                                    5  10   6
D000          3   F   1   3
D200              6D   A   A   1   9   B   8   D  16   5   E  11   9  30
D400      4        1   1   7   2   1   4   4   C   D  20  11   1   1   2
D600              6   9
D800                          1   2   5
DA00                                        8   F   9                   2
DC00                      3   7                   3   2
DE00                        42D1547               2
          0  20  40  60  80  A0  C0  E0 100 120 140 160 180 1A0 1C0 1E0
E000                      3   B   9   A
E200
E400
E600        838 E73
E800
EA00
EC00
EE00
F000
F200
F400
F600
F800
FA00
FC00
FE00
          0  20  40  60  80  A0  C0  E0 100 120 140 160 180 1A0 1C0 1E0

A>
```

End Listing One

Listing Two

Assembly Listing of the Program Monitor

Copyright © 1984 by Poor Person Software. All rights reserved.
Permission is granted for personal, non-commercial use only.

```
        aseg                    ; absolute assembly
false   equ     0
true    equ     not false
;
;       the following four equates must be changed for
;       the particular hardware configuration of your system
;
ccp     equ     0c700h          ; ccp base
intvct  equ     4               ; the S100 interrupt number of the clock
enable  equ     true            ; enable interrupts when done
nbits   equ     11              ; number of bits in counter address
;
```

(Continued on next page)

Program Monitor Package (Listing Continued, text begins on page 26)

Listing Two

```
;       11 bits means every 32 bytes
;       10 bits means every 64 bytes
;       9 bits means every 128 bytes
;       8 bits means every 256 bytes
;
bdos    equ     5               ; jump for bdos
boot    equ     0               ; warm boot place
space   equ     2 shl (nbits) + 0200h
pmpbas  equ     ccp-space       ; base of pmp when active
setdma  equ     26              ; bdos equate for setdma
rstdsk  equ     13              ; bdos equate for reset disk system
pstring equ     9               ; bdos equate for print string
fcb     equ     05ch            ; fcb base (for input options)
intloc  equ     intvct*8        ; location of jmp for interrupt
;
        org     0100h
;
        lda     fcb+2           ; f or n (part of "off" or "on")
        cpi     'N'             ; on
        jz      onpmp
        cpi     'F'             ; off
        jz      offpmp
        ret                     ; forget it
onpmp:
;
;       see if already on
;
        lxi     h,intrcpt       ; my bdos jump address
        lda     bdos+1          ; compare with bdos jump that is active
        cmp     l
        jnz     onit            ; it is not mine so we can take over
        lda     bdos+2
        cmp     h
        rz                      ; it is already on
onit:
;
;       move the code into high memory
;
        lxi     h,pmpbas        ; destination of code
        lxi     d,pmp           ; assembled code
        lxi     b,0200h         ; move this many
10:
        ldax    d               ; get byte
        mov     m,a             ; put byte
        inx     h
        inx     d
        dcx     b
        mov     a,b
        ora     a
        jnz     10
        mov     a,c
        ora     a
        jnz     10              ; move all bytes
;
;       now modify the existing addresses in 1 and 6
;
        di                      ; we must not be interrupted
        lhld    bdos+1          ; current bdos jump
        shld    jmpbd+1         ; save bdos jump so can restore
        lxi     h,intrcpt       ; get new jump to bdos
        shld    bdos+1          ; replace bdos jump so can intercept them
```

```
;
;       copy the disk error addresses
;
        mvi     b,8             ; this many bytes
        lhld    jmpbd+1
        xchg                    ; old bdos address to de
        lhld    bdos+1          ; new bdos address
        mvi     e,9
        mvi     l,9             ; offset of address list
cl1:
        ldax    d
        mov     m,a
        inx     h
        inx     d
        dcr     b
        jnz     cl1             ; copy all 8 bytes
;
;       intercept disk read/write routines to our place
;       so we can disable the clock in case the BIOS doesn't
;       mask interrupts when it needs to
;
        lhld    1               ; bios vector base
        lxi     b,37            ; offset for read
        dad     b               ; to 2 byte address of read
        mov     e,m
        inx     h
        mov     d,m             ; get both bytes
        xchg
        shld    biored          ; to our call
        xchg
        lxi     d,disred
        mov     m,d
        dcx     h
        mov     m,e             ; put in our address
;
        lhld    1               ; vector base
        lxi     b,40
        dad     b               ; to 2 byte address for write
        mov     e,m
        inx     h
        mov     d,m             ; to de
        xchg
        shld    biowrt          ; to our call
        xchg
        lxi     d,diswrt
        mov     m,d
        dcx     h
        mov     m,e             ; put ours into vector
;
;       now do the same for warm boot
;
        lhld    1               ; address of warm boot jump
        inx     h               ; low byte
        mov     e,m
        inx     h
        mov     d,m             ; get it
        xchg
        shld    oldjp0          ; save it
        xchg
        lxi     d,jmp0          ; our address
        mov     m,d
        dcx     h
        mov     m,e
;
        if      enable
        ei
        endif
```

(Continued on page 36)

Program Monitor Package (Listing Continued, text begins on page 26)

Listing Two

```
;
        ret
offpmp:
;
;       see if pmp is active
;
        lxi     h,intrcpt       ; my bdos intercept address
        lda     bdos+1
        cmp     1               ; see if we are on
        rnz                     ; no return quickly
        lda     bdos+2
        cmp     h
        rnz                     ; it ain't us
;
        di
        lda     active          ; is it now active
        ora     a
        jz      notact          ; no leave interupt alone
        lhld    oldtjp+1        ; restore old timer routine
        shld    intloc+1        ; if intloc was not a jump it wasnt meaningful
notact:
        lhld    jmpbd+1         ; restore old bdos jump address
        shld    bdos+1
;
;       must restore bios vectors
;
        lhld    oldjp0          ; warm boot
        xchg
        lhld    1
        inx     h               ; to correct place
        mov     m,e             ; low byte
        inx     h
        mov     m,d             ; high byte
;
        lhld    biored          ; disk read
        xchg
        lhld    1
        lxi     b,37
        dad     b
        mov     m,e
        inx     h
        mov     m,d             ; put back
;
        lhld    biowrt
        xchg
        lhld    1
        lxi     b,40
        dad     b
        mov     m,e
        inx     h
        mov     m,d
;
        lda     active          ; were we on
        ora     a
        jz      pcount          ; no don't fool with clock
        lda     clockon         ; see if clock was on to start
        ora     a               ; well
        jnz     pcount          ; yes (leave clock on)
        call    disclk          ; disarm clock
pcount:
```

```
        if      enable
        ei
        endif
        ret
;
        ds      64
stack:
;
;       from here on all labels must be of the form
;
;  L    EQU     $-PMP+PMPBAS
;
;       to cause the assembler to generate code that can be moved
;       to the location in memory (PMPBAS) where it will reside
;       when it is run.  There are other ways to do this with
;       other assemblers
;
pmp     equ     $               ; moved to high memory
;
intrcpt equ     $-pmp+pmpbas
;
;       must trap the reset system call
;
        jmp     start
        ds      14
start   equ     $-pmp+pmpbas
        mov     a,c             ; see if system reset
        ora     a
        jz      jmp0            ; yes simulate warm boot
        cpi     252             ; how about on
        jz      startc
        cpi     251             ; how about off
        jz      stopc
jmpbd   equ     $-pmp+pmpbas
        jmp     0               ; filled in with "real" bdos address
startc  equ     $-pmp+pmpbas
;
;       zero counters
;
        lda     active
        ora     a               ; is it active
        lxi     h,0             ; if doesn't work set "false"
        rnz                     ; yes cant start the started
bytes   equ     space-0200h
        lxi     b,bytes
        lxi     d,pccnt
12      equ     $-pmp+pmpbas
        xra     a               ; get a zero
        stax    d               ; put a zero
        inx     d
        dcx     b
        mov     a,b             ; check high byte
        ora     a
        jnz     12
        mov     a,c             ; check low byte
        ora     a
        jnz     12
;
;       now must "borrow" the timer interupt.  It may be on
;       or it may not be.  must determine the state of timer interrupts
;
        xra     a
        sta     clockon         ; clear flag
        lhld    intloc+1        ; possible timer addr
        shld    oldtjp+1        ; save old address (if active must be jmp)
        mvi     a,jmp           ; set up my interrupt
        sta     intloc
```

(Continued on next page)

Program Monitor Package (Listing Continued, text begins on page 26)

Listing Two

```
        lxi     h,timjp0        ; come here on interrupt
        shld    intloc+1
;
;       see if clock is on
;       this is done to determine if the previous timer interrupt
;       routine is active.  The presence of a jump instruction at
;       location intloc does not mean that the timer is running.  Our
;       first little timer routine will simply set a flag if it is
;       called
;
;
        ei
        lxi     d,0ff00h
13      equ     $-pmp+pmpbas
        nop
        nop
        nop
        dcx     d
        mov     a,d
        ora     a
        jnz     13              ; wait awhile
;
;       if clockon = 1 then clock was running
;
        di
        lxi     h,timjp1        ; bypass set of clockon
        shld    intloc+1        ; set up real interrupt routine
        mvi     a,1
        sta     active
        ei
;
        lda     clockon
        ora     a
        jz      onclok          ; must start clock
        xra     a
        lxi     h,1             ; set "true"
        ret
;
;       turn on clock because it was not active before
;
onclok  equ     $-pmp+pmpbas
        call    armclk          ; turn on clock
;
        xra     a               ; go go
        lxi     h,1             ; set "true"
        ret
;
stopc   equ     $-pmp+pmpbas
        lda     active          ; are we active
        ora     a
        lxi     h,0             ; set "false"
        jz      stopna          ; no
        di
        lhld    oldtjp+1        ; restore old timer routine
        shld    intloc+1        ; if intloc was not a jump it wasnt meaningful
        lhld    jmpbd+1         ; restore old bdos jump address
        shld    bdos+1
        lxi     h,pccnt         ; "true" in spades
        lda     clockon
        ora     a
```

```
        jnz     stopna          ; it was on leave it on
        call    disclk          ; off the clock
;
stopna  equ     $-pmp+pmpbas
;
        if      enable
        ei
        endif
;
        xra     a
        sta     active
        ret
clockon equ     $-pmp+pmpbas
        db      0               ; start with clock off
active  equ     $-pmp+pmpbas
        db      0               ; 1 if PMP active
;
;       read sector routine  must disable clock interrupts here
;
disred  equ     $-pmp+pmpbas
        push    psw
        call    disclk          ; disarm clock
        pop     psw
biored  equ     $-pmp+1+pmpbas
        call    $               ; bios read sector filled in
        push    psw
        call    armclk          ; arm clock
        pop     psw
        ret
;
;       write sector routine  must disable clock interrupts here
;
diswrt  equ     $-pmp+pmpbas
        push    psw
        call    disclk          ; disarm clock
        pop     psw
biowrt  equ     $-pmp+1+pmpbas
        call    $               ; bios write sector filled in
        push    psw
        call    armclk          ; arm clock
        pop     psw
        ret
;
;       put your arm clock routine here
;
armclk  equ     $-pmp+pmpbas
        lda     active
        ora     a
        rz                      ; not active is nop
        mvi     a,050h          ; reset clock flag
        out     6
        mvi     a,0c0h          ; arm interrupts
        out     6
        ei
        ret
;
;       put. your disarm clock routine here
;
disclk  equ     $-pmp+pmpbas
        lda     active
        ora     a
        rz                      ; not active  is nop
        mvi     a,040h
        out     6
        ret
;
jmp0    equ     $-pmp+pmpbas
```

(Continued on next page)

Program Monitor Package (Listing Continued, text begins on page 26)

Listing Two

```
        lxi     h,intrcpt       ; must reset this
        shld    bdos+1          ; because DDT not restore it when done
;
        lxi     sp,0100h        ; get default stack
        lxi     d,080h          ; set dma
        mvi     c,setdma
        call    bdos
        mvi     c,rstdsk        ; use bios reset disk
        call    bdos
        lda     4               ; get user number and disk
        mov     c,a             ; for ccp
        jmp     ccp+3           ; fake warm boot
;
;       must test to see if we need to arm the clock before
;       returning.  If there is another routine active allow
;       it the priviledge of arming the clock for the next
;       interrupt
;
retmyt  equ     $-pmp+pmpbas    ; come here at end of my timer routine
        push    psw
        lda     clockon         ; see if some one else active
        ora     a
        jnz     oldtj1          ; yes no need to on the clock
        call    armclk
        pop     psw
        ret
oldtj1  equ     $-pmp+pmpbas
        pop     psw
oldtjp  equ     $-pmp+pmpbas
        jmp     0               ; filled in
;
;
;       determine if the clock is on
;
;       this is a timer interrupt routine that is used to determine
;       if the clock is being used.  It sets a flag if called and
;       goes on to the next routine (if present)
;
timjp0  equ     $-pmp+pmpbas
        push    psw             ; if we get here during initialization
        mvi     a,1             ; the clock is on
        sta     clockon         ; say was on
        pop     psw
        jmp     oldtjp          ; go to the previous user of timer interrupt
;
;       the run-time timer interupt routine
;       increment the counter indicated by the PC
;
timjp1  equ     $-pmp+pmpbas
        shld    savehl          ; need to get PC at interupt time
        pop     h               ; get into h
        shld    pcint           ; PC at interrupt
        push    h               ; put it back
        push    psw             ; save state of machine
        mov     h,b
        mov     l,c
        shld    savebc
;
;       increment the correct counter
;
```

```
;       form counter number as high order n bits of the pc address
;
        lda     pcint           ; get low order interrupt address
        mov     c,a             ; to c
        lda     pcint+1         ; get high order address byte
        mov     b,a             ; to b
        ora     a               ; clear carry
;
;       form high n bits of address  via a double shift right
;       by 15-n   (this leaves address * 2 to address 2 byte
;       counters
;
;
shiftc  equ     15-nbits
;
;                               ; 4 for 11 bits (32 bytes)
        lxi     h,shiftc        ; 5 for 10 bits (64 bytes)
;                               ; 6 for  9 bits (128 bytes)
shiftl  equ     $-pmp+pmpbas
        mov     a,b             ; high byte
        rar                     ; low bit to carry 0 to high bit
        mov     b,a             ; put it back
        mov     a,c             ; low byte
        rar                     ; bring in bit from high, low to carry
        ani     0feh            ; zap low bit (times 2)
        mov     c,a             ; put it back
        dcr     l
        jnz     shiftl          ; do it n times
;
        lxi     h,pccnt         ; get base of counters
        dad     b               ; form the counter address
        mov     c,m             ; get low byte
        inx     h
        mov     b,m             ; get high byte
        inx     b               ; bump counter
        mov     m,b             ; put it back
        dcx     h
        mov     m,c             ; both back
        pop     psw
        lhld    savebc
        mov     b,h
        mov     c,l
        lhld    savehl          ; restore h
        jmp     retmyt          ; go to next processor or return
;
savehl  equ     $-pmp+pmpbas
        ds      2
savebc  equ     $-pmp+pmpbas
        ds      2
pcint   equ     $-pmp+pmpbas
        ds      2
oldjp0  equ     $-pmp+pmpbas
        ds      2
;
;       counters
;
nc      equ     1 shl (nbits)
        dw      nc
pccnt   equ     $-pmp+pmpbas
        end
```

End Listing Two

(Continued on next page)

Program Monitor Package (Listing Continued, text begins on page 26)

Listing Three

The C program used to control the Program Monitor and to format the counters.

Copyright © 1984 by Poor Person Software. All rights reserved.
Permission is granted for personal, non-commercial use only.

```
/*      This program interfaces with the Program Monitor Package
        to begin sampling, stop sampling, and format counter
        values for printing.  The PMP intercepts BDOS calls
        #251, reset counters and begin sampling, and #252
        stop sampling and return a pointer to sample counters
*/

#include bdscio.h

        char *fname;                        /*    pointer to file name */
        char iobuf[BUFSIZ];                 /*    I/O buffer area      */

main(argc,argv)
        int argc;                           /*    number of command tokens */
        char **argv;                        /*    list of pointers to tokens */
{
        int bdos();                         /*    declare function      */
        int *ptr;                           /*    declare a pointer     */
        char str[5];                        /*    a string for conversions  */
        int ncounter,addr,lctr,addrinc,i,iarg;   /* declare integers */
        char *optname;                      /*    pointer to option     */

        if (argc <2) {                      /*    first token is command */

/*      if no arguments (only command token) print help information  */

                printf("Usage:\n\nMONITOR fname -options");
                printf("\n\noptions:\n\n");
                printf("    -B => begin  (fname ignored)\n");
                printf("    -E => end  (Output to fname if present)\n");
                exit();
        }

        fname="";                           /* initialize pointer to null string */

        for (iarg = 1; iarg < argc; ++iarg) {   /* process arguments one at */
                                                /* a time in this loop      */

          if(*argv[iarg] == '-') {          /* first character of token is '-'  */
                                            /* must be an 'option'              */
             optname = argv[iarg] +1;       /* set pointer to option (passed)-  */

             if (!strcmp(optname,"B") || !strcmp(optname,"BEGIN")) {

                                            /* look for B or Begin              */
               if (bdos(252,0) == 0) {      /* call PMP to start samples        */
                  puts("Program monitor not installed or already active\n");
                  exit();
               }

             }
             else if (!strcmp(optname,"E") || !strcmp(optname,"END")) {

                                            /* option is E or END.  print       */
                                            /* results.  Open optional file     */
                                            /* for results.                     */

                if (*fname)                 /* a file has been specified        */
```

```
                if (fcreat(fname,iobuf) == ERROR) {
                    puts("File create error on %s\n",fname);
                    exit();          /* disk must be full!              */
                }
              addr=0;                /* initialize address value for    */
                                     /* each line                       */
              ptr=bdos(251,0);       /* stop sampling and get pointer   */
                                     /* to counters                     */
              if(ptr == 0) {         /* if zero function not present    */
                 printf("Program monitor not active\n");
                 if(*fname) fclose(iobuf);
                 exit();             /* exit                            */
              }
              ncounter=*(--ptr);     /* number of counters is in front  */
                                     /* of the counters                 */
              addrinc=ncounter/16;   /* 16 counters per line            */
                                     /* addrinc is now number of lines  */
              addrinc=(16384/addrinc)*4;  /* same as 65536/addrinc      */
                                     /* addrinc is address range covered*/
                                     /* by each line                    */
              lctr=0;                /* initialize the line counter     */
              ptr++;                 /* back to first counter value     */
              while (ncounter-- ) {     /* loop for each counter        */

                                     /* every 16  lines give a header   */
        if ((lctr % 256) == 0) header(addrinc);

                          /* print the address at the beginning */
                          /* of each line                       */

        if ((lctr++ % 16)== 0) {

            if (addr ) htos(addr,str);  /* convert address to */
            else strcpy(str,"   0");  /* hex, or special case */

                             /* if file specified output to it  */
                             /* else put to console             */

           if(!*fname) printf("\n%s  ",str);       /* no file */
            else fprintf(iobuf,"\n%s  ",str);      /* file    */
            addr +=   addrinc;          /* bump address value   */
        }
        htos(*ptr++,str);              /* convert counter to hex   */
        if (!*fname) puts(str);        /* no file put to console   */
        else fputs(str,iobuf);         /* file, put to file        */
      }                                /* loop all counters        */
      header(addrinc);                 /* trailing header          */
      if(*fname)  {                    /* if file then close       */
          putc('\032',iobuf);          /* properly with Z          */
          fclose(iobuf);
      }
   }

                                       /* unknown option           */
   else printf("Invalid option %s\n",optname);
}

        else fname = argv[iarg];              /* not an option, set file  */
                                              /* name pointer             */
      }
}
```

(Continued on next page)

Program Monitor Package (Listing Con*inued, text begins on page 26)

Listing Three

```
/*      put out a header
        the address increment for each line is divided by 16
        to set the address increment for each counter
*/

header(addrinc)
        int addrinc;
{
        int colinc,coladdr,i;                   /* some integers  */
        char str[5];                            /* a string for conversions */

        colinc=addrinc/16;              /* address increment for each column */
        coladdr=colinc;                 /* address over second column        */
        if(!*fname) puts("\n          0");      /*  first column to console */
        else fprintf(iobuf,"\n          0");    /*  first column to file    */
                                                /* loop for 15 more columns */
        for (i=1;i<16;i++) {
                htos(coladdr,str);              /* convert column address   */
                if (!*fname) puts(str);         /* put to console           */
                else fputs(str,iobuf);          /* or to file               */
                coladdr += colinc;          /* increment the column address */
        }
}

/*      hex to string conversion
        hex value is put into string with leading blanks
*/


htos(val,str)
        int val;
        char str[];
{
        int i;

        for (i=3;i>=0;i--)  {           /* i is nibble number from the right  */

                str[i] = (val >> ((3-i)*4))  15;        /* isolate nibble */

                if (str[i] < 10)        /* for decimal values offset by '0' */
                        str[i] += '0';
                else
                        str[i] += '7';                  /* '7' + 10 = 'A'   */
        }
        str[4]='\0';                                    /* terminate the string  */

        for (i=0; str[i] == '0'; str[i++] = ' '); /* replace leading zeros */
                                                  /* stopping at first digit */
}
```

End Listings

CP/M 2.2 Goes PUBlic

by Bridger Mitchell and Derek McKay

ZCPR, the widely used Z80 replacement for the standard Console Command Processor (CCP) supplied with CP/M 2.2, provides a much-appreciated search path feature. When you type a command (not prefixed by a drive designation), ZCPR first searches for the COM file on the logged-in drive in the current user number, next searches in the default user number (usually configured to user 0) on that drive, and finally looks in the default user number on drive A. The search terminates at the first matched filename; the command processor then loads that file and begins executing the program.[1]

This popular feature relieves you of the necessity of keeping track of where you have stored many programs. By putting commonly used programs in the default user number directory, you can make COM files accessible from all directories. In Rick Conn's ZCPR2 and ZCPR3, you can also specify a particular search path for any specific file; this capability allows you to construct hierarchical directories.

The ZCPR search path is implemented at the CP/M command level and therefore works only for COM files. Unfortunately, this limits its usefulness for programs that refer to other "associated" (non-COM) files. Many major programs—such as editors, formatters, data base managers, assemblers, and compilers—require overlays, work/swap files, libraries, dictionaries, and data base files that usually must reside in the directory of the logged-in drive and user number.

Currently three methods enable you to use such a program from more than one user number. You can use brute force to create copies of all of the files in several user numbers, a technique that very rapidly chews up disk storage. Or you can duplicate the directory entries (but not the files) into each user number with the public domain DUPUSR utility; this approach consumes more directory entries (but no additional disk space), and you run the risk of unintentionally deleting a key file should you forget to warm boot after erasing a duplicated directory entry. (After an erasure, the BDOS does not rebuild the disk group allocation vector and considers the "erased" sectors to be free for subsequent file-write operations.)

The third method is to load a search path utility, such as Michael Rubenstein's SETDRU, that intercepts BDOS disk function requests and redirects references to specified files. You first configure a filter utility with a list of associated files and their user numbers. The program (or a pre-program) then loads the filter into high memory where it traps and reinterprets requests for any associated file. SETDRU, however, uses some memory and cannot be run with other resident modules, such as EX and

"The PUBlic file approach requires no additional disk space, directory entries or memory; it leaves the original file intact; and it protects files against accidental erasure."

*Bridger Mitchell & Derek McKay, Plu*Perfect Systems, Box 1494, Idyllwild, California 92349.*

XSUB (batch processing programs—the former public domain, the latter from Digital Research), The Backgrounder®, and so on.

PUBlic Files

Plu*Perfect Systems has developed a new "PUBlic file" approach that is both simple and flexible. It requires no additional disk space, directory entries, or memory; it leaves the original files intact; and it protects files against accidental erasure. You simply set a single directory attribute bit—bit 2 of a filename—to make *any* file PUBlic and accessible from all user numbers on that drive.[2] For example, once an editor and its swap and overlay files have been marked PUBlic, you can edit a file stored in any user number by switching to that user number and running the editor. This requires only one copy and one directory entry of the editor and its associated files.

The key to the PUBlic file approach is a patch that causes the BDOS to match a requested filename, regardless of user number, if that filename in the disk directory has the second attribute bit set. Because the directory carries the status of the file, the BDOS can determine, in a single scan of the directory, which file to open.

Our initial tests of this approach were encouraging but revealed one troublesome side effect: wild card (ambiguous) filename requests such as ??.COM or *.OVL frequently would match a PUBlic file. This meant that DIR and most directory utilities would list all PUBlic files in every user number. Much worse, ERA*.* would unceremoniously wipe out every PUBlic file!

After some experimentation, we modified the BDOS patch so that a wild card request will never match a PUBlic filename. The only way to erase a PUBlic file now is to type its exact (unambiguous) filename; you can do this from any user number. This also means that PUBlic files are not normally displayed in directories, which gives them a special type of SYStem status. DIR will still display a PUBlic file—if you give its exact name. But because you are more likely to use DIR to determine what files you have, whether you can remember their names or not, something more is needed.

Enter the PUBLIC program: a supporting utility that (1) displays all PUBlic files, (2) lets you make any file PUBlic as long as it is the only file with that name in all user numbers, and (3) lets you return any PUBlic file to the private sector.[3] PUBLIC works because of a single exception to the "no wild card matching" rule: if the first (drive) byte of the file control block (fcb) is the CP/M wild card character "?", then the patched BDOS will match every directory entry, including PUBlic files and erased files. (The source code for PUBLIC is in Listing Two, page 61.) Thus PUBLIC can search every directory entry on the disk to display all filenames of interest. We have also added this feature to the popular super directory (SD) utility; the extended version shows PUBlic files in a separate listing category.

We considered, and rejected, the CP/M 3.0 and MP/M method of using the SYStem attribute bit to make a file PUBlic if it is in user 0. In addition to requiring longer patches not easily shoehorned into the BDOS, this approach is risky. Using STAT you can set the SYStem bit of a user 0 file, and if that file also exists in user *n*, the results can be disastrous: in servicing each extent of a file request logged into user *n*, the BDOS will open whichever entry first appears in the disk directory.

Patching the BDOS

We wrote the BDOS patch (Listing One, page 51) in Z80 code, which enabled us to fit it entirely within the BDOS. The functional changes are all in the internal BDOS function FINDNXT, which finds the next directory entry matching the supplied fcb. To gain space we recoded part of another routine GETNEXT and also used the free bytes at the end of the BDOS. Because this area occasionally is used for other patches—e.g., the Kelly Smith ARCHIVE utility—you should check your copy of the BDOS for possible conflicts. The flow of control is somewhat indirect; we solved the integer packing problem of fitting code fragments into rather tight, odd-length segments by trial and error! To install the patch on 8080 machines, you will need to rewrite the patch in 8080 opcodes and find some extra space in the BIOS.

One of the CDL MACRO Z80 assemblers or the Digital Research MAC is well suited for the patching task because it generates hex output for the exact code area without adding extra zero-filled bytes. After assembling the patch, use DDT to load it with an offset directly on top of the system image. If you don't have an assembler with this feature, you can assemble the patch, load it into an area above the SYSGEN image, and move the exact number of bytes into place with DDT.[4]

It goes without saying that you should thoroughly test any file system patching first on floppies and, of course, back up all files before beginning.

A CP/M 2.2 Bug

In working out the BDOS patch, we uncovered a CP/M bug: although the CP/M 2.2 Interface Guide states that the Rename and Erase functions return the values 0, 1, 2, or 3 on a successful operation, in fact CP/M 2.2 returns only 0 on success (and 0FFh on failure). Moreover, the Set Attribute function uses the same exit code, yet the Guide omits any mention of this function's return status value. It turns out that the correct code is actually shorter, and we have included it in our patch. It seems desirable to make this correction since a program that uses one of these functions may occasionally wish to access the BIOS directory buffer to retrieve the directory fcb for the matched entry.

Copying Files with Attribute Bits

Many programs have a preference for the private sector. In both the original CCP and the replacement ZCPR, the RENname file command removes all attributes. Similarly, most file copy utilities (e.g., PIP, SWEEP, and DISK7) create the copy with no attribute bits. Thus, copying a PUBlic file with PIP creates a private version; if the destination drive is the same as the source, this operation also erases the original PUBlic source file. Many editors also do not preserve attribute bits, so editing a PUBlic file turns it into a private citizen once again.

To support renaming and copying of PUBlic files (as well as SYStem and Read/Only files), we have extended Frank Gaude's DISK7. The new version copies a file with its attribute bits intact and has an option for listing all PUBlic files. In PUBLIC, SD, and

DISK7 we adopted the convention of displaying a filename character in lower case when its attribute bit is set. This convention readily identifies PUBlic, SYStem, and Read/Only files. It is almost totally portable, and programmers revising other file copy, directory, and communications programs might consider adopting it.

PUBlic files, combined with the ZCPR-type search path for loading programs, provide 90% of the control over user numbers that is needed on a CP/M system. What is still missing is the ability within most running programs to use private files in several user numbers. An excellent solution is to build file reference by user number into the language, as Leor Zollman has done in version 1.5 of his BDS-C. With this feature, for example, MINCE can concurrently edit files from several user numbers. Otherwise, you may be able to fix up major programs on a piecemeal basis. For example, we have patched the Perfect Writer version that was bundled with the Kaypro computers to add a "change user number" command.

The PUBlic file approach does differ in some respects from a fully implemented search path because private and PUBlic files with the same name cannot peacefully coexist on the same drive. Its principal advantages are that a single-pass search of the directory suffices for each open-file service request; the code occupies no extra space on a CP/M system and does not interfere with resident modules in high memory; and user programs needn't be modified.

We have found PUBlic files to be a major convenience. For example, we used a PUBlic editor and swap file in a single editing session to revise this article and the program files, which were stored in two separate user numbers, and to retrieve notes stored on a backup floppy from a third.

Availability

The PUBlic file BDOS patches and PUBLIC utility program are copyrighted © 1984 by Plu*Perfect Systems, all rights reserved. We are releasing them for single-user use, with the restriction that they be used solely for noncommercial purposes. We hope that they will prove useful to many others. No use of this code in commercial software or systems is permitted in any circumstances without advance written permission from Plu*Perfect Systems.

Plu*Perfect Systems markets an enhanced operating system for Kaypro computers that incorporates the PUBlic file and command search path features and adds bidirectional typing capability. For users with other hardware, we can supply a disk for noncommercial use with all of the PUBlic files (source code for the patches and the PUBLIC utility, copies of the extended SD.COM and DISK7.COM, an assembled hex-relocatable version of the patch, and a relocating loader) for $12 on either an 8-inch SSSD format disk or a number of 5¼-inch formats. Please specify the format you need.

Notes

1. Richard Conn, Ron Fowler, Keith Peterson, and Frank Wancho developed ZCPR and placed it in the public domain. It is available from SIG/M User Group, Box 97, Iselin, NJ 08830. ZCPR3, written solely by Rick Conn, is available from Echelon, Inc., 101 First St., Los Altos, CA 94022. Ben Goldfarb has adapted portions of the Z80 ZCPR code to create an 8080 version of the CCP that incorporates the search path feature.
2. We do not use attribute bit 1 because we have already used it as a loading flag for keyboard definitions files for The Backgrounder—a resident background process utility currently available for Kaypro computers that also allows keys to be redefined dynamically.
3. PUBLIC is convenient but not essential; any program, such as Rick Conn's PROTECT, that can set and reset all attribute bits will move files to and from the public sector. (Version 1.0 of PROTECT has one bug that causes it to fail to find large files on large hard disk systems because of an incorrectly calculated extent number.) Be careful not to set the PUBlic attribute bit of a file that has the same name as a file in another user number.
4. Alternatively you could assemble the patch for a 32K CP/M system and install it in the MOVCPM image. However, this would require generating and installing a new bitmap for the modified bytes. We find it easier to have separate system images for the handful of different system sizes needed for any one machine.

DDJ

CP/M 2.2 (Text begins on page 48)

Listing One

```
 2  .settim 21:30:00
 3  .setdat 04/13/84
 6  .remark ~
 7                          -- PUBPATCH --
 8
 9  A CP/M 2.2 BDOS modification to support the PUBlic filetype.
10
11          ------------------------------------------
12           Copyright (c) 1984 -- All rights reserved.
13
14                      Plu*Perfect Systems
15                        P. O. Box 1494
16                      Idyllwild CA 92349
17          ------------------------------------------
```

(Continued on next page)

```
18
19  Attribute bit 2 of a filename signifies a PUBlic file,
20  accessible by its unambiguous filename from all user
21  numbers.
22
23  PUBlic files are not accessible via the usual ambiguous
24  filenames (e.g. *.* or ABC.D?F), to prevent unintentional
25  erasure and avoid directory clutter.
26
27  Directory entries for PUBlic files are, however, accessible
28  via ambiguous filenames by using the BDOS search-for-first,
29  search-for-next functions with a '?' in the drive-byte of the
30  fcb.  Extended versions of SD and DISK7 displays PUBlic files.
31
32  To erase a PUBlic file, use "ERA unambiguous-filename". Or use
33  DISK7.  Or change it to a private file and then erase with a
34  wildcard erase command.
35
36  The PUBLIC.COM utility is available to make files either
37  PUBlic or private and to list the current PUBlic files.
38
39  If another utility is used to set the PUBlic attribute bit,
40  avoid creating multiple files with the same name on the same
41  drive, unless all of them are private. (PUBLIC.COM checks
42  for this situation and prevents a conflict.)
43
44  The RENname command removes all attributes, so RENaming a
45  PUBlic file will make it private, R/W, DIR in its original
46  user number.
47  ~

50  .remark ~
51                          --- TO INSTALL ---
52
53  1a. Determine the BIOSBASE address of your system in memory
54      by subtracting 3 from the warm-boot address in memory:
55          DDT
56          LØ
57          subtract 3 ==> BIOSBASE address
58  1b. Subtract 1600H to determine the CCPBASE_MEMORY address.
59  1c. Assemble PUBPATCH for these addresses.
60
61      Either: use CDL's MACROIII assembler:
62
63          MACROIII PUBPATCH A:DHK
64
65      Or: convert the pseudo-opcodes to your assembler's
66          pseudo-ops and assemble into a HEX file.
67          e.g.    .loc    ==>     org
68                  =       ==>     set
69                  =\      ==>     ????,   etc.
70
71  2. Create a system image for the SYSGEN operation.
72     There are two ways to get the image:
73
```

```
                74      a. Either use SYSGEN to extract a system image from a disk
                75         in the usual manner --
                76           SYSGEN
                77           source drive? A
                78           destination drive? <CR>
                79           SAVE pp ORIG.SYS.    Use pp=50 pages or so to get
                80                                   the entire BIOS.
                81
                82      b. Or generate a new system --
                83           MOVCPM ss *              where ss=64 for a 64K system,
                84           SAVE pp ORIG.SYS         or whatever you are running.
                85
                86    3. Find the base address of the Command Processor in the image
                87           DDT ORIG.SYS
                88      Look for the command processor at 980H:
                89      You recognize it by two JMP instructions, followed by
                90      the command buffer (containing a Digital Research
                91      copyright notice, in the case of the original CCP):
                92           L980
                93           D980
                94      Call that address CCPBASE_IMAGE (normally 980H).
                95      (If you don't find it, you have a non-standard system, and
                96      your user's manual should have a memory map.  See, e.g.,
                97      the Compupro Disk 1 Controller manual, sec. 6.4).
                98
                99    4. Calculate the offset needed to cause the PUBPATCH.HEX file to
               100       load on top of the BDOS image.  'offset' will satisfy:
               101
               102           CCPBASE_IMAGE = CCPBASE_MEMORY + offset
               103
               104    5. Create a new system image containing the patch:
               105
               106           DDT ORIG.SYS
               107           IPUBPATCH.HEX
               108           Roffset
               109           G0
               110           SAVE pp NEWSYS.SYS
               111
               112    6. Finally, put the new system on a FLOPPY disk for testing:
               113
               114           SYSGEN NEWSYS.SYS
               115           <CR>
               116           destination_drive
               117
               118
               119           ------------------------------------------------
               120           Code also corrects a CP/M 2.2 bug that caused
               121           Rename, Set Attribute, and Delete File functions
               122           to return 0 status on success instead of 0,1,2,3
               123           per CP/M 2.2 Installation Guide.
               124           ------------------------------------------------
               125   ~
               128           .phex
EA00           129   biosbase =\ "Enter BIOS base address for PUBPATCH (xxxxH) "
               130
DC00           131   bdosbase = biosbase - 0e00h
D400           132   ccpbase  = biosbase - 1600h
               133
               134           .remark ~
```

(Continued on next page)

CP/M 2.2 (Listing Continued, text begins on page 48)

Listing One

```
                    135  biosbase = 0ea00h ; kayprol0 v 1.9
                    136           = 0fa00h ; kaypro 2
                    137           = 0dc00h ; h89
                    138  ~
                    139  ;       Internal BDOS locations:
                    140  ;
E32D                141  FINDNXT =       bdosbase+072Dh
E205                142  NXENTRY =       bdosbase+0605h
E1F5                143  CKFILPOS =      bdosbase+05F5h
E17F                144  MOREFLS =       bdosbase+057Fh
E15E                145  FCB2HL  =       bdosbase+055Eh
E307                146  SAMEXT  =       bdosbase+0707h
E1FE                147  STFILPOS =      bdosbase+05FEh
DF01                148  SETSTAT =       bdosbase+0301h
E9D9                149  SAVEFCB =       bdosbase+0DD9h
E9D8                150  COUNTER =       bdosbase+0DD8h
E9EA                151  FILEPOS =       bdosbase+0DEAh
DF45                152  STATUS  =       bdosbase+0345h
E9D4                153  FNDSTAT =       bdosbase+0DD4h
E154                154  CHKWPRT =       bdosbase+0554h
E144                155  CHKROFL =       bdosbase+0544h
E318                156  FINDFST =       bdosbase+0718h
E26B                157  SETFILE =       bdosbase+066bh
E1C6                158  DIRWRITE =      bdosbase+05c6h
E1F5                159  CKFILPOS =      bdosbase+05f5h
E8D7                160  DELFILE =       bdosbase+0cd7h
E9C5                161  EXTMASK =       bdosbase+0dc5h
E9D2                162  CLOSEFLG =      bdosbase+0dd2h
E9D3                163  RDWRTFLG =      bdosbase+0dd3h
E524                164  GETEMPTY =      bdosbase+0924h
E45A                165  OPENIT1 =       bdosbase+085ah
E0BB                166  STRDATA =       bdosbase+04bbh
DF01                167  SETSTAT =       bdosbase+0301h
DF05                168  IOERR1  =       bdosbase+0305h
E178                169  SETS2B7 =       bdosbase+0578h
                    170  ;


                    173          .pabs
E32D                174          .loc    FINDNXT
                    175
E32D  21 0000       176  fnxt0:  lxi     h,0
E330  22 E399       177          shld    pflag   ;initialize PUBlic & wildcard flags
E333  4C            178          mov     c,h     ;0
E334  CD E205       179          call    NXENTRY
E337  CD E1F5       180          call    CKFILPOS
E33A  2866          181          jrz     nomatch         ;if done
E33C  2A E9D9       182          lhld    SAVEFCB
E33F  EB            183          xchg                    ;de=user-fcb
E340  1A            184          ldax    d
E341  FEE5          185          cpi     0E5h    ;if Getempty fn  wants first
E343  2807          186          jrz     fnxt1   ;..deleted file slot in directory
E345  D5            187          push    d
```

```
E346  CD E17F   188          call    MOREFLS
E349  D1        189          pop     d
E34A  3056      190          jrnc    nomatch         ;if no more files
              191 ;
E34C  CD E15E   192 fnxtl:   call    FCB2HL          ;hl=directory fcb
E34F  3A E9D8   193          lda     COUNTER
E352  47        194          mov     b,a             ;b=count
E353  0E00      195          mvi     c,0             ;c=byte #
E355  B7        196          ora     a               ;COUNTER=0 ==> Search fn
E356  2847      197          jrz     matched         ;..so match every entry
              198 ;
E358  79        199 fnxt2:   mov     a,c             ;get byte #
E359  FE0D      200          cpi     13
E35B  2836      201          jrz     nxtbyte         ;omit S1 byte
E35D  1A        202          ldax    d               ;get user-fcb char
E35E  FE3F      203          cpi     '?'
E360  2005      204          jrnz    fnxt3
E362  32 E39A   205          sta     qflag   ;flag wildcard
E365  182C      206          jr      nxtbyt
              207 ;
E367  7E        208 fnxt3:   mov     a,m     ;get directory-fcb char
E368  FEE5      209          cpi     0E5h    ;check for blank/deleted file
E36A  79        210          mov     a,c     ;A = byte #
E36B  2814      211          jrz     chkext  ;if a deleted file, omit user # check
E36D  B7        212          ora     a
E36E  2011      213          jrnz    chkext  ;or if not user # byte
E370  23        214          inx     h       ;else check for PUBlic file
E371  23        215          inx     h
E372  CB7E      216          bit     7,m     ;..at attribute bit 2
E374  2B        217          dcx     h
E375  2B        218          dcx     h
E376  2809      219          jrz     chkext  ;if not PUBlic, match on user #
              220 ;
              221 ; the file is PUBlic
              222 ; -- but is BDOS looking for an empty directory slot?
E378  78        223          mov     a,b     ;if COUNTER=1, this is a Getempty request

E379  3D        224          dcr     a       ;
E37A  28B1      225          jrz     fnxt0   ;..so go to next file
E37C  32 E399   226          sta     pflag   ;else flag the file PUBlic,
E37F  1812      227          jr      nxtbyt  ;..and omit matching user #
              228 ;
E381  FE0C      229 chkext:  cpi     12      ; (A=byte #)
E383  1A        230          ldax    d
E384  2805      231          jrz     tstext  ;extent byte(#12) is special case
E386  96        232          sub     m       ;compare the characters
E387  E67F      233          ani     07fh    ;..excluding attribute bits
E389  1806      234          jr      extdone
E38B  C5        235 tstext:  push    b       ;check for same extent
E38C  4E        236          mov     c,m
E38D  CD E307   237          call    SAMEXT
E390  C1        238          pop     b
E391  209A      239 extdone:jrnz     fnxt0   ;if mismatch, get next file
```

(Continued on next page)

CP/M 2.2 (Listing Continued, text begins on page 48)

Listing One

```
                    240  ;
E393  13            241  nxtbyt: inx     d       ;chars match, bump to next byte
E394  23            242          inx     h
E395  ØC            243          inr     c       ;byte # ++
E396  1ØCØ          244          djnz    fnxt2   ;count--
                    245  ;
                    246  ; here if-- COUNTER > 1 and filenames match
                    247  ;
                    248  ;Test for PUBlic file and wild-card combination:
                    249  ;                       ;flags initially = Ø, but
E399                250  pflag = .+1             ;  = COUNTER-1 if PUBlic
E39A                251  qflag = .+2             ;  = '?'(3Fh) if '?' in fcb+1...
E398  21 ØØØØ       252          lxi     h,-.
E39B  7D            253          mov     a,l             ;if file is PUBlic
E39C  A4            254          ana     h               ;..and there's a wildcard
                    255                                  ;(3Fh & 1...n) ==> NZ
E39D  2Ø8E          256          jrnz    fnxtØ           ;..get next directory entry
                    257  ;
                    258  ; here if--
                    259  ;     (a) non-PUBlic filenames match,
                    260  ;  or (b) find-all-files (searchfirst/searchnext functions
                    261  ;         with drive byte = '?')
                    262  ;  or (c) delete unambiguous PUBlic-filename.
                    263  ;
E39F  C3 E5AE       264  matched:jmp     PATCH1
                    265  ;
E3A2  CD E1FE       266  nomatch:call    STFILPOS
E3A5  7D            267          mov     a,l             ;1=Øffh
E3A6  C3 DFØ1       268          jmp     SETSTAT
                    269  ;

                    272  ;       the ERASE FILE routine -- in a new location
                    273  ;
                    274  ;       Routine is split, with remainder stuffed
                    275  ;       into free bytes at end of BDOS.
                    276  ;
E3A9  CD E154       277  ERAFILE:call    CHKWPRT ;write-protect aborts
E3AC  ØE ØC         278          mvi     c,12
E3AE  CD E318       279          call    FINDFST
E3B1  CD E1F5       280  erafl:  call    CKFILPOS ;check for 'E5' case
E3B4  C8            281          rz
E3B5  CD E144       282          call    CHKROFL ;read-only file aborts
E3B8  CD E15E       283          call    FCB2HL
E3BB  C3 E9EE       284          jmp     PATCH2
E3BE                285  LAST1 = .       ;must be <= bdosbase+Ø7BEh
                    286  ;
                    287  ;
                    288
                    289
                    290  ;Remainder of ERASE routine goes at end of BDOS
                    291  ;
E9EE                292          .loc    bdosbase+Ødeeh  ;there are 18 spare bytes
```

```
                    293
E9EE                294  PATCH2:
E9EE  36E5          295  eraf2:  mvi     m,0E5h  ;install erase mark
E9F0  0E00          296          mvi     c,0
E9F2  CD E26B       297          call    SETFILE ;clear file's space in bitmap
E9F5  CD E1C6       298          call    DIRWRITE ;write directory sector
E9F8  CD E32D       299  eraf3:  call    FINDNXT ;look for next entry
E9FB  C3 E3B1       300          jmp     eraf1
E9FE                301  LAST2 = .       ;must be <= bdosbase+0E00h
                    302  ;


                    305  ;rewrite last part of bdos GETNEXT routine to gain space
                    306  ;
E571                307          .LOC    bdosbase+0971h
                    308  ;
E571  2815          309  gnxt0:  jrz     gnxt1   ;( overlaying jz gtnext1)
E573  47            310          mov     b,a     ;extent byte
E574  3A E9C5       311          lda     EXTMASK
E577  A0            312          ana     b
E578  21 E9D2       313          lxi     h,CLOSEFLG
E57B  A6            314          ana     m
E57C  2812          315          jrz     gnxt2   ;must read next extent
E57E  CD E45A       316  gnxt3:  call    OPENIT1 ;open current extent
E581  CD E0BB       317  gnxt4:  call    STRDATA ;update rec#, extent#,...
E584  AF            318          xra     a
E585  C3 DF01       319  jsetst: jmp     SETSTAT
                    320  ;
                    321  ;       have overflowed normal extent, check s2 byte
E588  23            322  gnxt1:  inx     h       ;shorter code, replacing
E589  23            323          inx     h       ;lxi b,2 & dad b
E58A  34            324          inr     m       ;bump s2 byte
E58B  7E            325          mov     a,m
E58C  E60F          326          ani     0fh
E58E  2818          327          jrz     gnxt5   ;error if too many extents
E590  0E0F          328  gnxt2:  mvi     c,15    ;open the next extent
E592  CD E318       329          call    FINDFST
E595  CD E1F5       330          call    CKFILPOS
E598  20E4          331          jrnz    gnxt3
E59A  3A E9D3       332          lda     RDWRTFLG ;no extant extent
E59D  3C            333          inr     a       ;..if reading, can't open one
E59E  2808          334          jrz     gnxt5
E5A0  CD E524       335          call    GETEMPTY ;writing, so get next free entry
E5A3  CD E1F5       336          call    CKFILPOS
E5A6  20D9          337          jrnz    gnxt4   ;and if no error, save the data
E5A8  CD DF05       338  gnxt5:  call    IOERR1  ;set error &
E5AB  C3 E178       339          jmp     SETS2B7 ;..don't close file
                    340  ;
                    341  ;
                    342  ;       use space (14 bytes) for fragment from FINDNXT routine:
                    343  ;
E5AE  3A E9EA       344  PATCH1: lda     FILEPOS
E5B1  E603          345          ani     03h
```

(Continued on page 60)

CP/M 2.2 (Listing Continued, text begins on page 48)

Listing One

```
E5B3  32 DF45     346           sta     STATUS  ;save its directory buffer index
E5B6  21 E9D4     347           lxi     h,FNDSTAT
                  348   ;
                  349           .remark ~
                  350   The original CP/M 2.2 code removed below is erroneous, and
                  351   causes BDOS Erasefile, Renamefile, Setattribute functions
                  352   to return A=0 on success rather than the directory index
                  353   (0,1,2 or 3) specified in the Interface Guide.
                  354
                  355   ;;      mov     a,m
                  356   ;;      ral
                  357   ;;      rnc
                  358   ;;      xra     a
                  359   ~
                  360   ;
E5B9  77          361           mov     m,a     ;also save it for use
E5BA  C9          362           ret             ;..by Erase,Rename,Set Attribute fns
                  363   ;
E5BB              364   LAST3 = .       ;must be <= bdosbase+09BCh
                  365   ;
                  366   ;
                  367   ;patch ERAFILE reference to its new location
                  368   ;
E8DA              369           .loc    (DELFILE+3)
E8DA  CD E3A9     370           CALL    ERAFILE
                  371   ;
                  372           .end
```

```
+++++ Symbol Table +++++

BDOSBA DC00       BIOSBA EA00       CCPBAS D400       CHKEXT E381
CHKROF E144       CHKWPR E154       CKFILP E1F5       CLOSEF E9D2
COUNTE E9D8       DELFIL E8D7       DIRWRI E1C6       ERAF1  E3B1
ERAF2  E9EE       ERAF3  E9F8       ERAFIL E3A9       EXTDON E391
EXTMAS E9C5       FCB2HL E15E       FILEPO E9EA       FINDFS E318
FINDNX E32D       FNDSTA E9D4       FNXT0  E32D       FNXT1  E34C
FNXT2  E358       FNXT3  E367       GETEMP E524       GNXT0  E571
GNXT1  E588       GNXT2  E590       GNXT3  E57E       GNXT4  E581
GNXT5  E5A8       IOERR1 DF05       JSETST E585       LAST1  E3BE
LAST2  E9FE       LAST3  E5BB       MATCHE E39F       MOREFL E17F
NOMATC E3A2       NXENTR E205       NXTBYT E393       OPENIT E45A
PATCH1 E5AE       PATCH2 E9EE       PFLAG  E399       QFLAG  E39A
RDWRTF E9D3       SAMEXT E307       SAVEFC E9D9       SETFIL E26B
SETS2B E178       SETSTA DF01       STATUS DF45       STFILP E1FE
STRDAT E0BB       TSTEXT E38B
```

End Listing One

Listing Two

```
CP/M RMAC ASSEM 1.1

                        title 'PUBLIC.ASM 4-12-84 (c) 1984 Plu*Perfect Systems'
                        PAGE 55
                ;
000A =          vers    equ     1$0
                ;
                ;Utility to set/clear attribute bit 2  of filename.
                ;For use with Plu*Perfect Systems' PUBlic-file BDOS patch.
                ;
                ;usage: PUBLIC [d:]                     -- lists PUBlic files (on d:)
                ;       PUBLIC [d:]filename.typ         -- makes filename.typ PUBlic
                ;       PUBLIC [d:]filename.typ [X]     -- makes filename.typ private
                ;       filename,type must be unambiguous - no wildcards
                ;
                ;prints drive/user number of all files of specified name,
                ;  with attribute bits displayed as lower-case characters
                ;
                ;if just one such file found:
                ;       if X, clears attribute bit - making file private in orig user #
                ;       else,   sets attribute bit - making file PUBlic
                ;
                ;
                        maclib z80

0000 =          FALSE   EQU     0
FFFF =          TRUE    EQU     NOT FALSE
                ;
0000 =          LISTPUBBIT      equ     0
0001 =          MAKPUBBIT       equ     1
007C =          fence           equ     '|'
                ;
0000 =          NULL    equ     0
0007 =          BELL    equ     'G'-'@'
000D =          CR      equ     0dh
000A =          LF      equ     0ah
001B =          ESC     equ     1bh
0020 =          SPACE   EQU     ' '
007F =          DEL     EQU     7FH

000E =          drivefn equ     14
000F =          openfn  equ     15
0010 =          closefn equ     16
0011 =          srchfstfn equ   17
0012 =          srchnxtfn equ   18
0013 =          deletefn equ    19
0014 =          readfn  equ     20
0015 =          writefn equ     21
0016 =          makefn  equ     22
0017 =          renamefn equ    23
0018 =          logvecfn equ    24
0019 =          curdskfn equ    25
001A =          dmafn   equ     26
001E =          setattrfn equ   30
001F =          getaddrfn equ   31
```

(Continued on next page)

CP/M 2.2 (Listing Continued, text begins on page 48)

Listing Two

```
0020 =          userfn  equ     32
0021 =          readrfn equ     33
0022 =          writerfn equ    34
0023 =          sizefn  equ     35
                ;
000F =          reccnt  equ     15
0020 =          currec  equ     32
0021 =          r0      equ     33
0080 =          tbuff   equ     80h
005C =          fcb     equ     5ch
006C =          fcb2    equ     fcb+16
                ;
0000 =          BOOT equ        0000h
0005 =          BDOS equ        0005h
FFFF =          ADDRESS equ     0FFFFh  ;filled in by loader
0E00 =          bdoslen         equ     0e00h

                dobdos  macro   num,arg
                        if not nul arg
                        lxi     de,arg
                        endif
                        mvi     c,num
                        call    5
                        endm
                ;
                print   macro   msg
                        if not nul msg
                        lxi     d,msg
                        endif
                        call    printde
                        endm
                ;
                $-MACRO
                        PAGE
                        aseg
0100                    org 100h
                ;
0100 C38702     top:    jmp     start
                ;
0103 0D0A505542banner:  db      CR,LF,'PUBLIC v '
010E 312E30             db      vers/10+'0','.',(vers mod 10)+'0'
0111 2028632920         db      ' (c) 1984 Plu*Perfect Systems$'
                ;
012F 0D0A0A5573usage:   db      CR,LF,LF,'Usage:  '
013A 5055424C49         db      'PUBLIC [d:]          -- list PUBlic files (on d:)'
016C 0D0A202020         db      CR,LF,'       '
0176 5055424C49         db      'PUBLIC [d:]file.ext   -- make file PUBlic'
019F 0D0A202020         db      CR,LF,'       '
01A9 5055424C49         db      'PUBLIC [d:]file.ext X -- make file private'
01D3 0D0A0A24           db      CR,LF,LF,'$'
                ;
01D7 0D0A0A5468header:  db      CR,LF,LF,'The currently PUBlic files are:',CR,LF,'$'
                ;
```

```
01FC 2073657420tomsg:  db      ' set to ==> $'
0209 5055424C49pubnam: db      'PUBLIC   $'
0213 505249564privnam:db       'PRIVATE  $'
021D 20203D3D3Eismsg:  db      '  ==> is already $'
                ;
022F 0D0A072A2Amultimsg:db     CR,LF,BELL,'*** Multiple copies, can''t change!$'
0255 0D0A072A2Anonemsg:db      CR,LF,BELL,'*** No file!$'
0265 0D0A072A2Acantmsg:db      CR,LF,BELL,'*** Can''t make file $'
027D 0D0A284E6Fnopubs: db      CR,LF,'(None.)$'
                ;
                ;
0287            start:  sspd   ustack
028B 316505             lxi    sp,stack
028E                    print  banner
0294 215D00             lxi    h,fcb+1
0297 7E                 mov    a,m
0298 FE20               cpi    SPACE
029A 210D05             lxi    h,flags
029D                    res    LISTPUBBIT,m
029F                    jrnz   setup
02A1                    setb   LISTPUBBIT,m
02A3                    print  header
                ;
02A9            setup:  dobdos dmafn,buf       ;set directory buffer
02B1                    dobdos curdskfn        ;save user's drive
02B6 320F05             sta    drive
02B9 321105             sta    udrive
02BC 3A5C00             lda    fcb             ;check for specified drive
02BF B7                 ora    a
02C0                    jrz    savusr
02C2 3D                 dcr    a
02C3 320F05             sta    drive           ;login specified drive
02C6 5F                 mov    e,a
02C7                    dobdos drivefn,
                ;
02CC            savusr: dobdos userfn,0ffh     ;save user #
02D4 321205             sta    uuser
02D7                    dobdos getaddrfn       ;get extent mask for this drive
02DC 23232323           inx h ! inx h ! inx h ! inx h
02E0 7E                 mov    a,m
02E1 324704             sta    EXTMASK
02E4 AF                 xra    a
02E5 321005             sta    count
02E8 3A6D00             lda    fcb2+1
02EB FE58               cpi    'X'
02ED 210D05             lxi    h,flags
02F0                    res    MAKPUBBIT,m
02F2                    jrz    find
02F4                    setb   MAKPUBBIT,m
                ;
                        PAGE
                ;       find all filename entries in all user numbers
                ;
02F6 215C00     find:   lxi    h,fcb
02F9 363F               mvi    m,'?'           ;match ALL dir entries
02FB EB                 xchg
02FC                    dobdos srchfstfn,
0301 320E05             sta    indx            ;save position in buffer
0304 3C                 inr    a
```

(Continued on page 66)

Listing Two

```
0305                    jrz     done            ;no entries at all
                ;
0307 CD1604     findall:call    chknxt          ;is entry PUBlic or specified filename?
030A                    jrnz    findnxt
030C CD5304             call    savefcb         ;yes - save it
030F CD7304             call    setcol
0312 CD8D04             call    printentry      ; list it
0315 211005             lxi     h,count         ; and count it
0318 34                 inr     m
0319            findnxt:dobdos  srchnxtfn,fcb
0321 320E05             sta     indx
0324 3C                 inr     a
0325                    jrnz    findall
                ;
0327 210D05     nomore: lxi     h,flags
032A                    bit     LISTPUBBIT,m
032C 3A1005             lda     count
032F                    jrz     nom0
0331 B7                 ora     a
0332                    jrnz    done
0334                    print   nopubs
033A                    jr      done
                ;
033C D601       nom0:   sui     1
033E                    jrc     none
0340                    jrnz    nochanges
0342 210D05             lxi     h,flags         ;exactly 1 file found
0345                    bit     MAKPUBBIT,m
0347                    jrz     nom1
0349 CD9003             call    setpub
034C                    jr      done
034E CDC803     nom1:   call    setpriv
0351                    jr      done
                ;
                        PAGE
                nochanges:      ;can't be PUBlic if > 1 match on drive
0353                    print   multimsg
0359                    jr      done
                ;
035B            none:   print   nonemsg
0361                    jr      done
                ;
0363 110902     nopub:  lxi     d,pubnam
0366 CD0C04             call    cant
                ;fall thru
                ;
                ;       all done, restore drive/user and return
                ;
0369 210D05     done:   lxi     h,flags
036C                    bit     LISTPUBBIT,m
036E                    jrz     done0
0370                    print   usage
0376 3A1105     done0:  lda     udrive  ;relogin user's drive
0379 5F                 mov     e,a
```

```
037A                    dobdos  drivefn,
037F 3A1205     donel:  lda     uuser
0382 5F                 mov     e,a
0383                    dobdos  userfn,
0388 CDBF04             call    crlf
038B            xit:    lspd    ustack
038F C9                 ret
                ;
                ;       set the PUBlic attribute bit
                ;
0390 211705     setpub: lxi     h,pubfcb+2      ;test attr bit 2
0393                    bit     7,m
0395                    jrnz    ispub           ;quit if already PUBlic
0397                    setb    7,m             ;set attr bit 2
0399 3A1505             lda     pubfcb          ;save user # for output
039C 5F                 mov     e,a             ;set user # of this file
039D                    dobdos  userfn,
03A2 211505             lxi     h,pubfcb
03A5 3600               mvi     m,0             ;put default drive into fcb
03A7 EB                 xchg
03A8                    dobdos  setattrfn,
03AD 3C                 inr     a
03AE                    jrz     nopub
03B0                    print   tomsg
03B6                    print   pubnam
                setpend:
03BC 211505             lxi     h,pubfcb
03BF C39004             jmp     printentl
                ;
03C2 110902     ispub:  lxi     d,pubnam
03C5 C3FF03             jmp     istype
                ;
                ;
                ;       reset the PUBlic attribute bit
                ;
                setpriv:
03C8 3A1505             lda     pubfcb          ;save file user #
03CB 5F                 mov     e,a             ;set user # of fcb
03CC                    dobdos  userfn,
03D1 211705             lxi     h,pubfcb+2      ;reset PUBlic attr bit
03D4                    bit     7,m
03D6                    jrz     ispriv          ;quit if it's already private
03D8                    res     7,m
03DA 211505             lxi     h,pubfcb
03DD 3600               mvi     m,0             ;default drive
03DF EB                 xchg
03E0                    dobdos  setattrfn
03E5 3C                 inr     a
03E6                    jrz     nopriv
03E8                    print   tomsg
03EE                    print   privnam
03F4                    jr      setpend
                ;
                        PAGE
03F6 111302     nopriv: lxi     d,privnam
03F9 C30C04             jmp     cant
                ;
03FC 111302     ispriv: lxi     d,privnam
                ;       fall thru
```

(Continued on next page)

Listing Two

```
03FF D5         istype: push    d
0400                    print   ismsg
0406 D1                 pop     d
                ;       fall thru
                ;
                printde:                ;bdos string print function
0407 0E09               mvi     c,9
0409 C30500             jmp     5
                ;
040C D5         cant:   push    d
040D                    print   cantmsg
0413 D1                 pop     d               ;print 2nd msg
0414                    jr      printde
                ;
                ;       check next directory entry
                ;       if listing PUBlic files, ret Z if PUBlic and 0th extent
                ;       if matching a filename, ret Z if same name,type and extent
                ;       else ret NZ
0416 CD6304     chknxt: call    findentry
0419 7E                 mov     a,m
041A FEE5               cpi     0E5h            ;don't match erased entries
041C                    jrnz    chkn0
041E B7                 ora     a               ;set nz
041F C9                 ret
0420 23         chkn0:  inx     h
0421 3A0D05             lda     flags
0424                    bit     LISTPUBBIT,a
0426                    jrz     chkn1
                ;
                ;       list all PUBlic files
0428 23                 inx     h               ;point at 2nd char of filename
0429 7E                 mov     a,m
042A 2F                 cma
042B E680               ani     80h             ;check compl of attr bit
042D C0                 rnz                     ;not PUBlic - ret NZ
042E 110A00             lxi     d,12-2          ;have a PUBlic file,
0431 19                 dad     d               ;point at its extent byte
0432 AF                 xra     a               ;and check for extent 0
0433                    jr      chkn3
                ;
                ;       check for match with specified filename/extent 0
                ;note* doesn't  allow wild cards
0435 115D00     chkn1:  lxi     d,fcb+1
0438 060B               mvi     b,11            ;name & type
043A 1A         chkn2:  ldax    d
043B 96                 sub     m
043C E67F               ani     7fh             ;don't test attr bits
043E 23                 inx     h
043F 13                 inx     d
0440 C0                 rnz                     ;nz if no match
0441                    djnz    chkn2
0443 1A                 ldax    d               ;now check the extent
0444 4E         chkn3:  mov     c,m
```

```
                ;
                ;       check for same extent in A, C
                ;
                SAMEXT:                 ;z set if same extent
                ;;      push    b       ;;bc not needed
0445 F5                 push    psw
0447 =          EXTMASK equ $+1
0446 3E00               mvi     a,00h
0448 2F                 cma
0449 47                 mov     b,a     ;save mask
044A 79                 mov     a,c     ;mask C
044B A0                 ana     b
044C 4F                 mov     c,a     ;save in C
044D F1                 pop     psw     ;now do A
044E A0                 ana     b
044F 91                 sub     c
0450 E61F               ani     1Fh     ;check only legal bits 0-4
                ;;      pop     b
0452 C9                 ret
                ;
                ;
                savefcb:
0453 CD6304             call    findentry
0456 7E                 mov     a,m     ;save the user #
0457 329504             sta     fileuserno
045A 111505             lxi     d,pubfcb
045D 012000             lxi     b,32
0460                    ldir
0462 C9                 ret
                ;
                findentry:                      ;find entry in buf
0463 3A0E05             lda     indx            ;point to fcb found
0466 87                 add     a               ;*32
0467 87                 add     a
0468 87                 add     a
0469 87                 add     a
046A 87                 add     a
046B 216505             lxi     h,buf
046E 85                 add     l
046F 6F                 mov     l,a
0470 D0                 rnc
0471 24                 inr     h
0472 C9                 ret
                ;
0473 3A1005     setcol: lda     count
0476 E603               ani     03h
0478 CABF04             jz      crlf
047B CD8304             call    twosp
047E 0E7C               mvi     c,fence
0480 CDC604             call    charout
0483 0E20       twosp:  mvi     c,space
0485 CDC604             call    charout
0488 0E20               mvi     c,space
048A C3C604             jmp     charout
```

CP/M 2.2 (Listing Continued, text begins on page 48)

Listing Two

```
                printentry:             ;print drive/user/filename
048D CD6304             call    findentry
                printent1:                      ;print at (hl)
0490 E5                 push    h
0491 CDB704             call    printdrv
0495 =          fileuserno equ $+1
0494 3E00               mvi     a,00h
0496 CDD304             call    printuser
0499 E1                 pop     h
                ;fall thru
                print$fn:               ;print filename.ext
049A 23                 inx     h
049B 0608               mvi     b,8
049D CDA704             call    prfn
04A0 0E2E               mvi     c,'.'
04A2 CDC604             call    charout
04A5 0603               mvi     b,3
                ;fall thru
                prfn:                   ;print filename char, lowercase if attr bit set
04A7 7E                 mov     a,m
04A8 E67F               ani     7fh     ;kill attr bit when printing
04AA BE                 cmp     m
04AB                    jrz     prfn1
04AD F620               ori     20h     ;set lower case
04AF 4F         prfn1:  mov     c,a
04B0 CDC604             call    charout
04B3 23                 inx     h
04B4                    djnz    prfn
04B6 C9                 ret
                ;
                printdrv:               ;print drive
04B7 3A0F05             lda     drive
04BA C641               adi     'A'
04BC 4F                 mov     c,a
04BD                    jr      charout
                ;
04BF 0E0D       crlf:   mvi     c,CR
04C1 CDC604             call    charout
04C4 0E0A               mvi     c.LF
                ;fall thru
                ;
                charout:                ;preserve registers
04C6 E5                 push    h
04C7 C5                 push    b
04C8 D5                 push    d
04C9 59                 mov     e,c
04CA 0E02               mvi     c,2
04CC CD0500             call    5
04CF D1                 pop     d
04D0 C1                 pop     b
04D1 E1                 pop     h
04D2 C9                 ret
                ;
                printuser:              ;print A as user #
```

```
04D3 FE0A               cpi     10
04D5                    jrnc    printul
04D7 F5                 push    psw             ;1 space if sgl digit
04D8 0E20               mvi     c,' '
04DA CDC604             call    charout
04DD F1                 pop     psw
04DE 6F         printul:mov     l,a
04DF 2600               mvi     h,0
04E1 CDE804             call    printdec
04E4 0E3A               mvi     c,':'
04E6                    jr      charout
                ;
                printdec:
04E8 F5         DECOUT: PUSH    PSW             ;prints hl in decimal
04E9 C5                 PUSH    B
04EA D5                 PUSH    D
04EB E5                 PUSH    H
04EC 01F6FF             LXI     B,-10
04EF 11FFFF             LXI     D,-1
04F2 09         DECOU2: DAD     B
04F3 13                 INX     D
04F4 DAF204             JC      DECOU2
04F7 0110A00            LXI     B,10
04FA 09                 DAD     B
04FB EB                 XCHG
04FC 7C                 MOV     A,H
04FD B5                 ORA     L
04FE C4E804             CNZ     DECOUT  ;recursive
0501 7B                 MOV     A,E
0502 C630               ADI     '0'
0504 4F                 mov     c,a
0505 CDC604             call    charout
0508 E1                 POP     H
0509 D1                 POP     D
050A C1                 POP     B
050B F1                 POP     PSW
050C C9                 RET
                ;
050D 00         flags:  db      0
050E 00         indx:   db      0
050F 00         drive:  db      0
0510 00         count:  db      0
0511 00         udrive: db      0               ;user's drive
0512 00         uuser:  db      0
0513 0000       ustack: dw      0
0515 =          pubfcb equ $
0565 =          stack equ pubfcb +32 + 48
                ;
0565 =          buf     equ     stack
0515                    END
```

End Listings

A Guide to Resources for the C Programmer

by Terry A. Ward

"This guide to C language resources lists the books available on the language, the articles written concerning C, the producers of C compilers and software, and even an abridged guide to public domain C software."

Furniture: Chippendale
Automobiles: '57 Chevy
Authors: Cain, Hendrix, and Ream

The current rage among the game-playing public is "trivia" games. If we imagined the above three answers to be part of a trivia game, we could probably reconstruct the question of interest. Obviously, "Chippendale" and "'57 Chevy" refer to classic items of furniture and automobiles, respectively. But who or what are Cain, Hendrix, and Ream, and where do they fit in?

Faithful readers of *Dr. Dobb's* will recognize the names and perhaps come to the correct conclusion that they are "classic" authors of public domain and other significant C programs. Namely, they are the authors of a Small-C compiler (Cain), an improved Small-C compiler (Hendrix), and two full-screen text editors (Ream).

To many, though, the names of Cain, Hendrix, and Ream are a mystery. This article is a guide to such C language resources. It lists books available on the language, articles written concerning C, software producers of C compilers, and even an abridged guide to public domain C software.

Prior to compiling this guide, I wrote an article for *Byte* magazine (August 1983) that annotated 100 C articles and books. In the interim, this list has grown to over 200 article citations. The information presented below is available in expanded form in my forthcoming book *Applied Programming Techniques in C* to be published by Scott, Foresman and Company at the end of 1984. The information is current as of September 1984.

The Classics

Our first section of material deals with what might be called the classics of C programming. Just as a real programmer can write a Fortran program in any language, all true C programmers should be familiar with these classics.

On the book front, Kernighan and Ritchie's book *The C Programming Language* (Prentice-Hall, 1978) stands at the pinnacle, defining the very language. Equally classic is *The C Puzzle Book* by Alan C. Feuer. While not a guide to the language, Feuer's book nevertheless provides a novel approach to language understanding. Feuer presents segments of C code (some elegant, some atrocious) and asks "what does it do?" In the process of untangling his puzzles, one learns the C language. For the advanced programmer there are two books of applied techniques: my book from Scott, Foresman and the book by Purdum, Leslie, and Stegemoller.

On the program front are several articles that should be read by all C programmers. The work by Cain created the concept of Small-C. Hendrix has expanded the Small-C compiler and brought it up to date in his work. Finally, Ed Ream

---

Terry A. Ward, Academic Computing Services, University of Northern Iowa, Cedar Falls, IA 50614.

Portions of this material are excerpted from Applied Programming Techniques in C *by Terry A. Ward, published by Scott, Foresman and Company, © 1984, 1900 East Lake Avenue, Glenview, IL 60025.*

has created not one but two excellent screen editors for the C programmer. These prime examples of C software all first appeared in the pages of *Dr. Dobb's Journal.*

Books

The books listed below are all valuable to the C programmer. The works by Plum concerning standards and the Bell Laboratory materials are perhaps the most specialized. With the increasing popularity of C, the majority of these books are available in the mass-market bookstores that carry computer-related books and magazines. Those noted by an asterisk (*) are perhaps best suited for the neophyte C programmer.

Bell Laboratories. *Unix Programmer's Manual.* 2 volumes. New York: Holt, Rinehart and Winston, 1983.

*Chirlian, Paul M. *Introduction to C.* Beaverton, OR: Matrix Publishers, Inc., 1984.

Feuer, Alan R. *The C Puzzle Book.* Englewood Cliffs, NJ: Prentice-Hall, 1982.

*Hancock, Les, and Morris Krieger. *The C Primer.* New York: Byte/McGraw-Hill, 1982.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

*Kochan, Stephen G. *Programming in C.* Rochelle Park, NJ: Hayden Book Company, 1983.

Plum, Thomas. *C Programming Standards and Guidelines: Version U (UNIX and Offspring).* Cardiff, NJ: Plum Hall, 1982.

Plum, Thomas. *C Programming Standards and Guidelines: Version W (Whitesmith Version).* Cardiff, NJ: Plum Hall, 1982.

Plum, Thomas. *Learning to Program in C.* Cardiff, NJ: Plum Hall, 1983.

Purdum, Jack. *C Programming Guide.* Indianapolis, IN: Que Corporation, 1983.

Purdum, J., T. Leslie, and A. Stegemoller. *C Programmer's Library.* Indianapolis, IN: Que Corporation, 1983.

*Traister, Robert J. *Programming in C for the Microcomputer User.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

Zahn, C. T. *C Notes: A Guide to the C Programming Language.* New York: Yourdon Press, 1979.

Periodicals/Newsletters

The major periodical sources for C materials are *Dr. Dobb's Journal* and The C Users' Group newsletter publication. Readers of *Dr. Dobb's* are aware that the software in C that has appeared in these pages ranges from C compilers to sophisticated full-screen text editors. The latter group is a source of an informative newsletter and over 15 megabytes of C source code (as of June 1984).

The C Letter. Whitesmith's, Ltd., 97 Lowell Road, Concord, MA 01742.

C Users' Group. C Users' Group, 415 E. Euclid, McPherson, KS 67460.

Dr. Dobb's Journal. M & T Publishing, 2464 Embarcadero Way, Palo Alto, CA 94303.

Microsystems. Ziff Davis Publishing Company, One Park Ave., New York, NY 10016.

Unique. InfoPro Systems, P. O. Box 33, East Hanover, NJ 07936.

Unir Project. Unir Corporation, 5987 E. 71st, Suite 106, Indianapolis, IN 46220.

Unix Review. 2711 76th Avenue, Southeast, Mercer Island, WA 98040.

UNIX/World. Tech Valley Publishing, 289 S. San Antonio Road, Los Altos, CA 94022.

USENIX Assoc. P. O. Box 7, El Cerrito, CA 94530.

/usr/ Group. 4655 Old Ironside Dr., Santa Clara, CA 95054.

World Unix & C. P. O. Box 5314, Mt. Carmel, CT 06518.

C Compilers

The list below categorizes the available C compilers for microcomputers as of September 1984. The addresses for the companies listed are presented alphabetically at the end of this section:

CP/M compilers

Alcor Systems
Alpha Omega Computer Systems
BD Software
Carousel Microtools
The Code Works
Ecosoft
Elfin Systems
ISE-USA
J. E. Hendrix
Kadak Products Limited
Knowology
LSI Japan Co. Ltd.
Manx Software
New Generation Systems
Software Toolworks
Supersoft
Telecon Systems
Western Wares
Whitesmith's

Tandy TRSDOS compilers

Alcor
ISE-USA

IBM PCDOS/ MSDOS/ CP/M-86

Carousel Microtools
Computer Innovations
Control-C Software
Coriolis Company
c-Systems
C Ware
Datalight
Digital Research
Ecosoft
LanTech Systems
Lifeboat Associates
Manx Software
Mark Williams Co.
Microsoft Corp.
Software Kinetics, Ltd.
Supersoft

Telecon
Unisource Corporation
Whitesmith's

Apple DOS
Manx Software

Unix-like Software Tools
Carousel Microtools
J. E. Hendrix
Knowology
Lifeboat Associates
Mark Williams Co.
New Generation Systems
Whitesmith's

Compiler Addresses
Alcor Systems, 800 W. Garland Avenue, Suite 204, Garland, TX 75040.
Alpha Omega Computer Systems, P. O. Box U, Corvallis, OR 97339.
BD Software, P. O. Box 2368, Cambridge, MA 02238.
Carousel Microtools, 609 Kearney Street, El Cerrito, CA 94530.
The Code Works, 5266 Hollister, Suite 224, Santa Barbara, CA 93111.
Computer Innovations, 980 Shrewsbury Avenue, Suite J504, Tinton Falls, NJ 07724.
Control-C Software, 6441 S.W. Canyon Court, Portland, OR 97221.
The Coriolis Company, P. O. Box 76, Clinton Corners, NY 12514.
c-Systems, P. O. Box 3253, Fullerton, CA 92634.
C Ware, P. O. Box 710097, San Jose, CA 95171.
Datalight, 11557 8th Avenue NE, Seattle, WA 98125.
Digital Research, P. O. Box 579, Pacific Grove, CA 93950.
Ecosoft, 6413 N. College Avenue, Indianapolis, IN 46220.
Elfin Systems, 265 Nogal Drive, Santa Barbara, CA 93110.
InfoSoft, 25 Sylvian Road, South Westport, CT 06880.
Introl Corp., 647 W. Virginia Street, Milwaukee, WI 53204.
ISE-USA, 85 W. Algonquin Road, Suite 400, Arlington Heights, IL 60005.
J. E. Hendrix, Box 8378, University, MS 38677-8378.
Kadak Products Ltd., 206-1847 W. Broadway Avenue, Vancouver, B.C., Canada V6J 1Y5.
Knowology, P. O. Box 283, Wilsonville, OR 97070.
LanTech Systems, 9635 Wendell Road, Dallas, TX 75243.
Lifeboat Associates, 1651 Third Avenue, New York, NY 10028.
LSI Japan Co. Ltd., 2-24-9 Yoyogi, Shibuya-Ku, Tokyo (151), Japan.
Manx Software, Box 55, Shrewsbury, NJ 07701.
Mark Williams Co., 1430 W. Wrightwood Avenue, Chicago, IL 60614.
Microsoft Corp., 10700 Northrup Way, Bellevue, WA 98004.
New Generation Systems, 1800 Michael Faraday Drive, Suite 206, Reston, VA 22090.
Programmer's Shop, 128-M Rockland Street, Hanover, MA 02339.
Software Kinetics, Ltd., 3 Amberwood Crescent, Nepean, Ontario K2E 7L1, Canada.
Software Toolworks, 15233 Ventura Blvd., Suite 1118, Sherman Oaks, CA 91403.
Supersoft, P. O. Box 1628, Champaign, IL 61820.
Telecon Systems, 1155 Meridian Avenue, Suite 218, San Jose, CA 95125.
tiny-C Associates, P. O. Box 269, Holmdel, NJ 07733.
Unipress Software, 2025 Lincoln Highway, Suite 312, Edison, NJ 08817.
Unisource Software, 71 Bent Street, Cambridge, MA 02141.
Vandata, 17544 Midvale Avenue North, Suite 107, Seattle, WA 98133.
Western Wares, Box C, Norwood, CO 81423.
Whitesmith's Ltd., 97 Lowell Road, Concord, MA 01742.

C Software

The firms listed below are the major resources for public domain (or relatively inexpensive non-public-domain) C programs and utilities that include source code.

Algorithmic Technology, Inc., P. O. Box 278, Exton, PA 19341-0278.
Blaise Computing, 2034 Blake Street, Berkeley, CA 94704.
C Source, 12801 Frost Road, Kansas City, MO 64138.
C Users' Group, 415 E. Euclid, McPherson, KS 67460.
Dedicated Micro Systems, 112 N. Main, Box 287, Yates Center, KS 66783.
Dr. Dobb's Journal, M & T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303.
Greenleaf Software, 2101 Hickory Road, Carrollton, TX 75006.
J. E. Hendrix, Box 8378, University, MS 38677-8378.
JMI Software Consultants, 1422 Easton Road, Roslyn, PA 19001.
Programmer's Shop, 128-M Rockland Street, Hanover, MA 02339.
Que Corporation, 7960 Castleway Drive, Indianapolis, IN 46250.
Ed Ream, 1850 Summit Avenue, Madison, WI 53705.
Software Toolworks, 15233 Ventura Blvd., Suite 1118, Sherman Oaks, CA 91403.
XOR Corporation, 5421 Opportunity Court, Minnetonka, MN 55343.
Western Wares Software, Box C, Norwood, CO 81423.

C Articles

Listed below are articles and other items from periodicals that relate to C programming, software tools, and utilities written in C. The index provides a quick guide to the basic topics that may be found within the articles.

001 Allison, Dennis. "A Monthly Algorithm Column." *Dr. Dobb's Journal*, 44: 44 – 45 (January 1980).
002 Allison, Dennis. "A Monthly Algorithm Column." *Dr. Dobb's Journal*, 46: 47 – 48 (June/July 1980).
003 Allison, Dennis. "A Monthly Algorithm Column." *Dr. Dobb's Journal*, 48: 44 – 45 (September 1980).
004 Anderson, Bruce. "Type Syntax in the Language C: An

Object Lesson in Syntactic Innovation." *ACM SIGPLAN Notices*, 15(3): 21 – 27 (March 1980).

005 Anderson, Gordon E., and Kenneth C. Shumate. "Selecting a Programming Language, Compiler and Support Environment: Method and Example." *IEEE Computer*, 29 – 36 (August 1982).

006 Anonymous. "Review of the C Programming Language (Kernighan & Ritchie)." *Computer Languages*, 4: 199 – 200 (1979).

007 Anonymous. "Bell Labs' 32 Bit C/UNIX Micro." *Pipes and Filters*, 1(1): 4 (June 1981).

008 Anonymous. "The C Programming Language." *Mini-Micro Software*, 6: n.p. (1981).

009 Ashcraft, Steven E. "Ultra Low Level Programming Using a High Level Language." In *Microcomputer Research and Applications: Proceedings of the First Conference of the HP/1000 International Users Group*, Helen K. Brown, ed., 168 – 184 (Elmsford, NY: Pergamon, 1981).

010 Azlin, Lawrence A. "A DEBUG Subroutine: A Technique for Making Program Debugging Simpler." *Microsystems*, 100 (December 1983).

011 Bailes, P. A. C. "A Co-routine Package for C." *Australian Computer Science Communications*, 1(4): 306 – 309 (December 1979).

012 Bailey, Kirk. "Small-C: Bug-Fix Bug." *Dr. Dobb's Journal*, 57: 4 (July 1981).

013 Bairstow, Jeffrey. "Getting Started with a New Language." *Personal Computing*, 7(12): 250 (December 1983).

014 Baker, Leslie, and Nat Sakowski. "New Improved Lattice C." *PC Magazine*, 138 – 141 (March 20, 1984).

015 Baker, Leslie, and Nat Sakowski. "Getting Your C-Legs." *PC Magazine*, 118 – 123 (March 20, 1984).

016 Barach, David R., and David H. Taenzer. "A Technique for Finding Storage Allocation Errors in C Language Programs." *ACM SIGPLAN Notices*, 17(5): 16 – 23 (May 1982).

017 Barach, David R., and David H. Taenzer. "A Technique for Finding Storage Allocation Errors in C Language Programs." *ACM SIGPLAN Notices*, 17(7): 32 – 38 (July 1982).

018 Barry, Steve, and Randy Jacobsen. "The TRS-80 Model 16B with Xenix." *Byte*, 288 – 320 (January 1984).

019 Bates, Dan. "I Can C Forever." *Dr. Dobb's Journal*, 67: 6 – 7 (May 1982).

020 Berenbaum, Alan D., Michael W. Condry, and Priscilla M. Lu. "The Operating System and Language Support Features of the BELLMAC-32 Microprocessor." *ACM SIGPLAN Notices*, 17(4): 30 – 38 (April 1982).

021 Bergmann, Ernest E. "PISTOL: A Forth-like Portably Implemented STack Oriented Language." *Dr. Dobb's Journal*, 76: 12 – 15 (February 1983).

022 Birman, H. K., L. N. Rolnitzky, and J. R. Biggee. "A Shape Oriented System for Holter ECG Analysis." In *Computers in Cardiology* (1978).

023 Black, Rodney. "Oh Say, Can I C!" *Dr. Dobb's Journal*, 61: 4, 51 (November 1981).

024 Bolstad, Terje. "CP/M BDOS and BIOS Calls for C." *Dr. Dobb's Journal*, 80: 22 – 27 (June 1983).

025 Bolton, Bill. "Some Useful C Time Functions." *Dr. Dobb's Journal*, 6(8): 16 – 21 (August 1981).

026 Bourne, Stephen J. "The Unix Shell." *Byte*, 187 – 204 (October 1983).

027 Boyd, Stowe. "Modular C." *ACM SIGPLAN Notices*, 18(4): 48 – 54 (April 1983).

028 Brooker, R. A. "A Database Subsystem for BCPL." *Computer Journal*, 25(4): 448 – 464.

029 Budd, Timothy C. "An Implementation of Generators in C." *Computer Languages*, 7: 68 – 87 (1982).

030 Burkowski, F. J., W. F. Mackey, and M. H. Hamza. "Micro-C: A Universal High Level Language for Microcomputers." *Proceedings of the IEEE International Symposium on Mini and Micro Computers* (Canada/USA, 1977/1978).

031 *Byte* (August 1983). The C Language (topical language issue).

032 Cain, Ron. "A Small-C Compiler for the 8080's." *Dr. Dobb's Journal*, 45: 5 – 46 (May 1980).

033 Cain, Ron. "Runtime Library for the Small-C Compiler." *Dr. Dobb's Journal*, 48: 4 – 15 (September 1980).

034 Cain, Ron. "High Praise for Small C." *Dr. Dobb's Journal*, 76: 10 (February 1983).

035 Calhoun, Herb. "Malpractice?" *Dr. Dobb's Journal*, 60: 4 – 5 (October 1981).

036 Cameron, A.G.W. "The Software Tools Computing Environment." *Microsystems*, 58 – 63 (September 1983).

037 Cameron, A.G.W. "The Ratfor Preprocessing Language." *Microsystems*, 52 – 56 (September 1983).

038 Cann, Peter. "C Source-Code Formatting." *Byte*, 18 (December 1983).

039 Cherry, Lorinda L., and Nina H. Macdonald. "The Unix Writer's Workbench Software." *Byte* (October 1983).

040 Christensen, Ward. "The CP/M Users Group Volume 48: Catalogue & Abstracts." *Lifelines*, 1(10): 15 – 16 (March 1981).

041 Christensen, Ward. "Full Screen Program Editors for CP/M-80: MINCE." *Lifelines*, 2(11): 7 – 11 (April 1982).

042 Christensen, Ward. "Full Screen Program Editors for CP/M-80: Ed Ream's Editor in C." *Lifelines*, 3(5): 43 – 45 (October 1982).

043 Christensen, Ward. "MINCE Revisited." *Lifelines*, 3(5): 45 (October 1982).

044 Clapp, Douglas. "Microsoft C Unveiled." *PC Magazine*, 503 – 508 (October 1983).

045 Clark, David D. "Lmodem: A Small Remote-Communication Program." *Byte*, 410 – 428 (November 1983).

046 Colley, William C. "6800 and 1802 Cross-Assemblers for CP/M." *Dr. Dobb's Journal*, 50: 38 – 39 (December 1980).

047 Colstad, Ken. "A High-Level Language for Easy X.25 Updates." *Data Communications*, 65 – 77 (September 1981).

048 Colvin, James L. "Small-C Compiler." *Dr. Dobb's Journal*, 52: 7, 37 (February 1981).

049 Cortesi, D. E. "Dr. Dobb's Clinic." *Dr. Dobb's Journal*, 11 – 19 (November 1983).

050 Cotton, G. "A Master Disk Directory." *Interface Age*, 6(11): 104 – 105, 162 – 167 (November 1981).

051 Currie, Edward H. "Form over Substance . . ." *Lifelines* (April 1983).
052 Daneliuk, Tim. "LC: A C Compiler for LDOS-Based Machines." *InfoWorld*, 5(25): 53 – 54.
053 Darwin, Ian F. "The Unix File." *Microsystems*, 24 – 27 (September 1983).
054 Datapro. "An Introduction to the 'C' Language." In *Applications Software Directory* (Delran, NJ: Datapro Research, 1980).
055 Dessey, Raymond, ed. "Languages for the Laboratory, Part I." *Analytical Chemistry*, 55(6): 650A – 662A (May 1983).
056 Dessey, Raymond, ed. "Languages for the Laboratory, Part II." *Analytical Chemistry*, 55(7): 754A – 764A (June 1983).
057 Dietz, Paul F. "Don't Knock C." *Byte*, 576 – 557 (November 1983).
058 Ditzel, David R., and H. R. McLellan. "Register Allocation for Free: The C Machine Stack Cache." *ACM SIGPLAN Notices*, 17(4): 48 – 55 (April 1982).
059 Dobyns, Barry A. "MINCE: Not Just Another Editor." *Dr. Dobb's Journal*, 6(4): 48 – 52 (April 1981).
060 Dobyns, Barry A. "Mincemeat and Minced Words." *Dr. Dobb's Journal*, 56: 4 (June 1981).
061 Duncan, Ray. "Tiny-C Interpreter on CDOS." *Dr. Dobb's Journal*, 35: 37 – 39 (September 1979).
062 Elliott, Comal. "A Very General Problem-Oriented CAI System." *Behavior Research Methods and Instrumentation*, 14(2): 165 – 169 (April 1982).
063 Emerson, Sandra L. "Usenet: A Bulletin Board for Unix Users." *Byte*, 219 – 236 (October 1983).
064 Favitta, Michael. "Unica and XM-80." *Dr. Dobb's Journal*, 78: 83 – 85 (April 1983).
065 Feuer, Alan R., and Narain H. Gehani. "A Comparison of the Programming Languages C and Pascal." *ACM Computing Surveys*, 14(1): 73 – 92 (March 1982).
066 Fiedler, David. "The BDS C Compiler." In *Programmer's Guide to CP/M*, Sol Libes, ed. (Morris Plains, NJ: Creative Computing Press, 1982).
067 Fiedler, David. "Unix Facilities on CP/M: Microshell." *Microsystems*, 51 – 53 (January 1983).
068 Fiedler, David. "The Unix Tutorial. Part 1: An Introduction to Features and Facilities." *Byte*, 186 – 219 (August 1983).
069 Fiedler, David. "The Unix Tutorial. Part 2 : Unix as an Applications-Programs Base." *Byte*, 257 – 278 (September 1983).
070 Fiedler, David. "The Supersoft C Compiler." *Microsystems* (September 1983).
071 Fiedler, David. "The Unix Tutorial. Part 3: Unix in the Microcomputer Marketplace." *Byte* (October 1983).
072 Fishman, Harvey. "Janus – A New Ada Compiler for Z80 Systems." *Microsystems*, (6): 70 – 81 (November/December 1982).
073 Fitzhorn, Patrick A., and Geatold R. Johnson. "C: Toward a Concise Syntactic Definition." *ACM SIGPLAN Notices*, 16(12): 14 – 21 (December 1981).
074 Fitzhorn, Patrick A., and Geatold R. Johnson. "C: Toward a Concise Syntactic Definition: Appendix." *ACM SIGPLAN Notices*, 17(8): 89 – 95 (August 1982).
075 Foulk, Richard. "Standard Deviation." *Dr. Dobb's Journal*, 77: 11, 90 (March 1983).
076 Garrett, Roger C. "Structured English for the C Programmer." *Interface Age*, 6(10): 30 – 34 (October 1981).
077 Garrett, Roger C. "More on C Programming." *Interface Age*, 6(11): 26 – 28, 158 (November 1981).
078 Garrett, Roger C. "C.Plus (Conclusion)." *Interface Age*, 6(12): 34 – 38, 142 – 143 (December 1981).
079 Gewirtz, David A. "An Introduction to the C Programming Language." *Microsystems*, 2(6): 20 – 38 (November/December 1981).
080 Gewirtz, David A. "An Introduction to the C Programming Language (Part II)." *Microsystems*, 3(1): 50 – 58 (January/February 1982).
081 Gewirtz, David A. "A Reply to Larry Hamelin." *Microsystems*, 3(4): 12 (July/August 1982).
082 Gewirtz, David A. "Two More C Compilers: A Comparative Review of the Aztec C II and C/80 Compiler." *Microsystems*, 83 – 89 (November/December 1982).
083 Gibson, T. A., and S. B. Guthery. "Structured Programming, C and tiny-C." *Dr. Dobb's Journal*, 5(5): 30 – 33 (May 1980).
084 Gilbreath, Jim. "A High-Level Language Benchmark." *Byte*, 6(9): 180 – 198 (September 1981).
085 Gilbreath, Jim, and Gary Gilbreath. "Erasthosthenes Revisited: Once More through the Sieve." *Byte*, 8(1): 283 – 326 (January 1983).
086 Gore, Mike, and Bernie Roehl. "Small-C Bug-Fix Bug." *Dr. Dobb's Journal*, 57: 4 – 6, 8 (July 1981).
087 Gorman, Walter, and Michael Groussard. "Minicomputer Programming Languages." *ACM SIGPLAN Notices*, 11(4): 4 – 15 (April 1, 1976).
088 Hagler, David. "Supersoft Speaks Up." *Byte*, 20 (February 1984).
089 Halfant, Matthew. "Small-C for the 9900." *Dr. Dobb's Journal*, 69: 66 – 71 (July 1982).
090 Halfant, Matthew. "The Unix C Compiler in a CP/M Environment." *Byte*, 243 – 267 (August 1983).
091 Hamelin, Larry. "Intro to the C Programming Language." *Microsystems*, (4): 12 (July/August 1982).
092 Hancock, Les. "Growing, Pruning and Climbing Binary Trees with tiny-C." *Dr. Dobb's Journal*, 4(5): 37 – 41, 54 (May 1979).
093 Hancock, L. "Implementing a Tiny Interpreter with a CP/M-flavored C." *Dr. Dobb's Journal*, 5(1): 20 – 28 (January 1980).
094 Hare, Van Court. "Learning C Inexpensively." *Lifelines*, 30 – 34 (December 1983).
095 Harrell, John B. "So Much to C." *80 Micro*, 100 – 106 (February 1984).
096 Hatch, William E. "Letter to the Editor." *Microsystems*, 22 (December 1983).
097 Hendrix, James, and Ernest Payne. "A New Library for Small-C." *Dr. Dobb's Journal*, 50 – 81 (May 1984) and 56 – 69 (June 1984).
097a Hendrix, J. E. "Small VM: Nucleus of a Portable Software Development Environment." *Dr. Dobb's Journal*, 61: 34 – 45 (November 1981).
098 Hendrix, J. E. "Small-C Expression Analyzer." *Dr.*

Dobb's Journal, 6(12): 40 – 43 (December 1981).
099 Hendrix, J. E. "Small Shell: Part 2 of a North Star VOS." *Dr. Dobb's Journal*, 63: 27 – 35 (January 1982).
100 Hendrix, J. E. "Small-C Compiler, V. 2." *Dr. Dobb's Journal*, 74: 16 – 52 (December 1982).
101 Hendrix, J. E. "Small-C Compiler, V. 2 Continued." *Dr. Dobb's Journal*, 75: 48 – 64 (January 1983).
102 Hendrix, J. E. "Small-C Notes, Bug Fix." *Dr. Dobb's Journal* (January 1983).
103 Hendrix, J. E. "A Slightly Slicker Fix for Small-C." *Dr. Dobb's Journal*, 77: 6 (March 1983).
104 Hinsch, Hanno. "Five C Language Compilers." *PC Magazine*, 210 – 216 (February 1983).
105 Hogan, W. L. "An Evaluation of a Raster Scan Display for Use in an Aircraft Information Handling System." Master's thesis, Naval Postgraduate School, Monterey, CA, 1977.
106 Houston, Jerry, Jim Broderick, and Les Kent. "Comparing C Compilers for CP/M-86." *Byte*, 82 – 106 (August 1983).
107 Houston, Jerry. "Reply to 'Supersoft Speaks Up'." *Byte*, 20 – 21 (February 1984).
108 Howard, Alan D. "Enhancing the C Screen Editor." *Dr. Dobb's Journal*, 79: 38 – 63 (May 1983).
109 Howard, J. E. "An Implementation of a Codasyl Based Data Base Management System under the Unix Operating System." Master's thesis, Naval Postgraduate School, Monterey, CA, 1978.
110 Hughes, Phil. "BASIC, Pascal, or tiny-C? A Simple Benchmarking Comparison." *Byte*, 6(10): 372 – 375 (October 1981).
111 Hunt, Bill. "C and the PC: Part 1." *PC Tech Journal*, 110 – 130 (November/December 1983).
112 Hunt, Bill. "C and the PC: Part 2." *PC Tech Journal*, 91 – 124 (January 1984).
113 Hunter, Bruce. "A Comparison of C Compilers." *Lifelines*, 3(12): 15 – 21 (May 1983).
114 Hunter, Bruce N. "A Review of Microshell—A Unix Like Utility." *Lifelines*, 3(11): 17 – 21 (April 1983).
115 Jackson, T. R. "Letter to the Editor." *Microsystems*, 3(5): 22 – 23 (September/October 1982).
116 Jaeschke, Rex. "Let's C Now . . . Part 1, In the Beginning . . ." *The DEC Professional*, 42 – 50 (January 1984).
117 Jaeschke, Rex. "Let's C Now . . . Part 2, Looping and Testing." *The DEC Professional*, 26 – 38 (March 1984).
118 Jalics, Paul J., and Thomas S. Heines. "Transporting a Portable Operating System: Unix to an IBM Minicomputer." *Communications of the ACM*, 26(12): 1066 – 1072 (December 1983).
119 Johannson, Jan-Henrik. "Argc and Argv for Small-C." *Dr. Dobb's Journal*, 74: 62 – 64 (December 1982).
120 Johnson, S. C. "A Portable Compiler: Theory and Practice." In *Proceedings 5th ACM Conference on Principles of Programming Languages* (1978).
121 Johnson, S. C., and D. M. Ritchie. "Portability of C Programs and the Unix System." *Bell System Technical Journal*, 2021 – 2048 (July/August 1978).
122 Johnson, Stephen C., and Brian W. Kernighan. "The C Language and Models of Systems Programming." *Byte*, 48 – 60 (August 1983).
123 Jolly, George W. "Review of *BCPL—The Language and Its Compiler* (Martin Richards and Colin Whitby-Strevens)." *Dr. Dobb's Journal*, 79: 70 (May 1983).
124 Joyce, J. "Review of *The C Puzzle Book*." *ACM Computing Reviews*, 286 (June 1982).
125 Joyce, James. "A C Language Primer. Part 1: Constructs and Conventions in C." *Byte*, 64 – 78 (August 1983).
126 Joyce, James. "A C Language Primer. Part 2: Tool Building in C." *Byte*, 289 – 302 (September 1983).
127 Joyce, James. "A Tour Through the Unix File System." *Byte*, 170 – 182 (October 1983).
128 Katzenelson, Jacob. "Introduction to Enhanced C (EC)." *Software Practice and Experience*, 13(7): 551 – 576 (July 1983).
129 Katzenelson, Jacob. "Higher Level Programming and Data Abstractions—A Case Study Using Enhanced C." *Software Practice and Experience*, 13(7): 577 – 595 (July 1983).
130 Kern, Christopher. "A User's Look at Tiny-C." *Byte*, 4(12): 196 – 206 (December 1979).
131 Kern, Christopher. "Printf for the C Function Library." *Byte*, 6(5): 40 – 434 (May 1981).
132 Kern, Christopher. "The BDS C Compiler." *Byte*, 6(6): 356 – 362 (June 1981).
133 Kern, Christopher. "MINCE: A Text Editor." *Byte*, 150 – 160 (September 1981).
134 Kern, Christopher. "Microshell and Unica: Unix-Style Enhancements for CP/M." *Byte*, 206 – 219 (December 1982).
135 Kern, Christopher. "Supervyz and Organizr: Two Menu-Driven Front Ends for CP/M." *Byte*, 446 – 451 (January 1983).
136 Kern, Christopher. "The Scribble Text Processor." *Byte*, 302 – 310 (February 1983).
137 Kern, Christopher O. "Five C Compilers for CP/M-80." *Byte*, 110 – 130 (August 1983).
138 Kern, Christopher. "More Unix-Style Software Tools for CP/M." *Byte*, 428 – 434 (October 1983).
139 Kernighan, B. "Why Pascal Is not my Favorite Programming Language." *Computer Science Technical Report #100* (Bell Laboratories, July 18, 1981).
140 King, B. "The Flexibility of C." *CP/M Review*, 1(2): 22 – 23, 75 (January/February 1983).
141 Krieger, M. S., and P. J. Plauger. "C Language's Grip on Hardware Makes Sense for Small Computers." *Electronics* (May 8, 1980).
142 Krieger, Mark, and Fred Pack. "Unix as an Application Environment." *Byte*, 209 – 214 (October 1983).
143 Kvaleberg, Egil. "Small-C DISKDOC: A Repair and Maintenance Utility." *Dr. Dobb's Journal*, 66: 26 – 34 (April 1982).
144 Kvaleberg, Egil. "Doctoring DISKDOC." *Dr. Dobb's Journal*, 67: 6 – 7 (May 1982).
145 Leas, Dennis, and Paul Wintz. "Chisel Your Code with a Profiler." *Byte*, 286 – 290 (August 1983).
146 Lee, P. A. "Exception Handling in C Programs." *Software Practice and Experience*, 13(1983): 389 – 405.
147 Libes, Don. "Reply to T. R. Jackson." *Microsystems*, 3(5): 23 – 26 (September/October 1982).

148 Libes, Don. "Unix and CP/M." *Microsystems*, 26 – 34 (January 1983).
149 Libes, Don. "Unix on Microcomputers." *Microsystems*, 42 – 44 (January 1983).
150 Linhart, Jason. "Managing Software Development with C." *Byte*, 172 – 182 (August 1983).
151 MacPherson, Andrew. "Catching Bugs in Small-C." *Dr. Dobb's Journal*, 81: 6 – 7 (July 1983).
152 Madden, J. Gregory. "C: A Language for Microprocessors?" *Byte*, 2(10): 130 – 138 (October 1977).
153 Magenat-Thalmann, Nadia. "Choosing an Implementation Language for Automatic Translation." *Computer Languages*, 7(1982): 161 – 170.
154 Mark of the Unicorn. MINCE text editor documentation. (Mark of the Unicorn, 1981).
155 Mateti, Prabhaker. "Pascal Versus C: A Subjective Comparison." In *Proceedings of the Symposium on Language Design and Programming Methodology*, 37 – 69 (Sydney, September 10 – 11, 1979).
156 Matthews, M. M. "Review: The C Primer." *ACM Computing Reviews*, 24(9): 384 (September 1983).
157 McCall, Caddy. "Microshell and Microtools." *CP/M Review*, 26 – 35 (November/December 1983).
158 McClure, Robert L. "Likes to C It Standard, Too." *Dr. Dobb's Journal*, 79: 7 (May 1983).
158a McDermott, Edward. "Optimizing Strings in C." *Dr. Dobb's Journal*, 18 – 23 (April 1984).
159 McKeon, Brian. "A Small-C Operating System." *Dr. Dobb's Journal*, 77: 36 – 61 (March 1983).
160 McSkimin, J. R. "REDAS—A Relational Data Access System for Real-Time Applications." In *Proceedings of COMPSAC 1978* (Computer Software and Applications Conference, 1978).
161 Meissner, Michael. "Letter to the Editor: Reply to Fitzhorn and Johnson." *ACM SIGPLAN Notices*, 17(8): 84 – 88 (August 1982).
162 Mohler, Lorin S. "A Disk Alignment Routine." *Microsystems*, 2(6): 70 (November/December 1981).
163 Murphy, Walter V. "RECLAIM: A File Reclamation Utility for Destroyed Directories." *Dr. Dobb's Journal*, 78: 14 – 23 (April 1983).
164 Ness, David, and A. Krigman. "MINCE Editor from Mark of the Unicorn." *Infoworld* (May 11, 1981).
165 Norris, Bill. "C-Bits (All About BDS C Version 1.45)." *Lifelines*, 2(9): 37 – 38 (February 1982).
166 Nowell, Scott. "Cross Check." *Microsystems*, 98 – 102 (September 1983).
167 Petersen, Holger. "Review of the BDS C Compiler." *Dr. Dobb's Journal*, 47: 49 – 50 (September 1980).
168 Phraner, Ralph A. "Nine C Compilers for the IBM PC." *Byte*, 134 – 168 (August 1983).
169 Plauger, P. J. "Review of *The C Programming Language*." *ACM Computing Reviews*, 2 – 4 (January 1979).
170 Pournelle, Jerry. "User's Column . . . MINCE is Not Complete EMACS . . ." *Byte*, 294, 298, 300 (July 1982).
171 Pournelle, Jerry. "User's Column . . . Kernighan's Lament . . . There's a New C A'Comin . . ." *Byte*, 226 – 236 (December 1982).
172 Pournelle, Jerry. "User's Column . . . There's a New C A'Comin . . ." *Byte*, 230, 235, 236 (December 1982).
173 Pournelle, Jerry. "The Debate Goes On . . ." *Byte*, 324 (August 1983).
174 Pournelle, Jerry. "User's Column . . . BDS C." *Byte*, 88 – 89 (January 1984).
175 Pugh, T. "MCALL-C: A Communications Protocol for Personal Computers." *Dr. Dobb's Journal*, 5(2): 16 – 20 (June/July 1980).
176 Pugh, T. "BDS C, A Full Compiler from Lifeboat Associates." *Infoworld* (March 31, 1980).
177 Ream, Edward K. "A Portable C Screen-Oriented Editor." *Dr. Dobb's Journal*, 63: 18 – 61 (January 1982).
178 Ream, Edward K. "Screen-Oriented Bugs." *Dr. Dobb's Journal*, 67: 4 – 7 (May 1982).
179 Ream, Edward K. "RED: A Better C Screen Editor, Part I." *Dr. Dobb's Journal*, 81: 34 – 65 (July 1983).
180 Ream, Edward. "RED: A Better C Screen Editor, Part II." *Dr. Dobb's Journal*, 82: 62 – 97 (August 1983).
181 Reed, Adam. "An Underline Filter for Matrix Printers." *Byte*, 7(3): 300 – 306 (March 1982).
182 Reid, Larry, and Andrew P. McKinlay. "Whitesmith's C Compiler." *Byte*, 8(1): 330 – 344 (January 1983).
183 Reitz, Randy. "Small-VOS and Small-Tools." *Microsystems*, 4(1): 66 – 69 (January 1983).
184 Rifkin, Edward M., and Steve Williams. "The C Language: Key to Portability." *Computer Design*, 143 – 150 (August 1983).
185 Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan. "The C Programming Language." *Bell System Technical Journal*, 57(6): 1991 – 2019 (July/August 1978).
186 Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan. "The C Programming Language." *Dr. Dobb's Journal*, 5(5): 20 – 29 (May 1980).
187 Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan. "The C Programming Language." Reprinted from *Bell System Technical Journal* (July/August 1978). In *Programming Languages: A Grand Tour*, Ellis Horwitz, ed. (Computer Science Press, 1983).
188 Roberts, Bruce. "C." *Popular Computing*, 111 – 119 (September 1983).
189 Robertson, M. D. "An Extended BASIC Compiler with Graphics Interface for the PDP-11/50 Computer." Master's thesis, Naval Postgraduate College, Monterey, CA, 1977.
190 Roth, Richard. "Small C Grows Up." *Dr. Dobb's Journal*, 61: 46 – 50 (November 1981).
191 Roth, Richard L. "C Language Prints Multiple Text Lines." *Electronics Design News*, 171 (August 4, 1982).
192 Rovegno, H. D. "Using C Language for Microprocessors." *Electro/77 Conference Record* (1977).
193 Rovegno, H. D. "A Support Environment for MAC-8 Systems." *Bell Systems Technical Journal*, 57(6; pt. 2): 2251 – 2264 (July/August 1978).
194 Runyon, John. "Review of C Notes: A Guide to the C Programming Language." *DEC Professional*, 22 – 26 (November 1982).
195 Saloman, F. A. "Software Development for Microprocessors—A Case Study." *Proceedings of COMPSAC 1978* (Computer Software and Applications Confer-

ence, 1978).
196 Scherrer, Deborah, K. Philip, H. Scherrer, Thomas H. Strong, and Samuel J. Penny. "The Software Tools: Unix Capabilities on Non-Unix Systems." *Byte*, 430 – 446 (November 1983).
197 Schreiner, Axel T. "*cc*—A driver for a Small-C Programming System." *Dr. Dobb's Journal*, 40 – 55 (June 1984).
198 Schreiner, Axel T. "*p*—A Small-C Preprocessor." *Dr. Dobb's Journal*, 46 – 82 (July 1984).
199 Sethi, Ravi. "A Case Study in Specifying the Semantics of a Programming Language." In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, 117 – 130 (Las Vegas, NV, January 28 – 30, 1980) (ACM, 1980).
200 Skjellum, Anthony. "Argum—A C Command Line Processor." *Dr. Dobb's Journal*, 10 – 31 (August 1982).
201 Skjellum, Anthony. "Using C Instead of Assembly Language." *Microsystems*, 33 – 36 (September/October 1982).
202 Skjellum, Anthony. "Expanding Wildcards Under Unix." *Dr. Dobb's Journal*, 73: 12 – 16, 43 (November 1982).
203 Skjellum, Anthony. "UNICA: A Unix-like Utility System for CP/M." *Microsystems*, 59 – 56 (January 1983).
204 Skjellum, Anthony. "C/Unix Programmer's Notebook." *Dr. Dobb's Journal*, 14 – 17 (December 1983).
205 Skjellum, Anthony. "C/Unix Programmer's Notebook." *Dr. Dobb's Journal*, 94 – 97 (February 1984).
206 Skjellum, Anthony. "C/Unix Programmer's Notebook." *Dr. Dobb's Journal*, 94 – 96 (April 1984).
207 Skjellum, Anthony. "C/Unix Programmer's Notebook." *Dr. Dobb's Journal*, 86 – 106 (June 1984).
208 Skjellum, Anthony. "C/Unix Programmer's Notebook." *Dr. Dobb's Journal*, 116 – 120 (August 1984).
209 Skjellum, Anthony. "C/Unix Programmer's Notebook." *Dr. Dobb's Journal*, 116 – 122 (September 1984).
210 Springer, Allen. "A Comparison of Language C and Pascal." *Technical Report G320-2128* (IBM Cambridge Scientific Center, Cambridge, MA, 1979).
211 Stallings, S. "C into the Future." *PC Magazine*, 142 – 147 (March 20, 1984).
212 Stankowski, J. B. "The Design and Implementation of a General Purpose Graphics Subroutine Library." Master's thesis, Naval Postgraduate College, Monterey, CA, 1976.
213 Staneff, John E. "A Small-C Concordance Generator." *Dr. Dobb's Journal*, 86 – 105 (August 1984).
214 Staneff, John. "A Small-C Help Facility." *Dr. Dobb's Journal*, 84: 40 – 69 (October 1983).
215 Stroustrup, Bjarne. "Classes: An Abstract Data Type Facility for the C Language." *ACM SIGPLAN Notices*, 7(1): 42 – 51 (January 1982).
216 Stroustrup, Bjarne. "Adding Classes to the C Language." *Software Practice and Experience*, 13(1983): 139 – 161.
217 Suckow, Harry. "We'll C You and Raise You . . ." *Dr. Dobb's Journal*, 8 – 9, 12 – 16 (January 1984).
218 Taylor, Jeff. "LIST—A Source-Listing Program for the C Language." *Byte*, 6(6): 234 – 246 (June 1981).
219 Taylor, Jeffrey L. "Cross-Reference Generator in C: A Program Conversion Aid." *Dr. Dobb's Journal*, 68: 50 – 54 (June 1982).
220 Taylor, Jeffrey L. "CREF Update." *Dr. Dobb's Journal*, 79: 7, 9 (May 1983).
221 Terry, Chris. "MINCE—A New Text Editor." *Microsystems*, 76 – 79 (May/June 1982).
222 Terry, Chris. "Book Review: *The C Primer, Learning to Program in C*, and *C Programming Guide*." *Microsystems*, 120 (September 1983).
223 Thomas, Rebecca. "What is a Software Tool?" *Byte*, 222 – 238 (August 1983).
224 Tilson, Michael. "Moving Unix to New Machines." *Byte*, 266 – 276 (October 1983).
225 Torkildson, Chris L. "Micro Compiler in Brobdingnag." *Dr. Dobb's Journal*, 72: 6 (October 1982).
226 Tuthill, Bill. "Typesetting on the Unix System." *Byte*, 253 – 262 (October 1983).
227 Van Zandt, James R. "Micro Compiler in Brobdingnag." *Dr. Dobb's Journal*, 72: 6, 59 – 60 (October 1982).
228 Watson, Ron. "Learning the C Language with C-System's C Compiler." *Lifelines*, 4(6): 7 – 8 (November 1983).
229 Watson, Ron. "Software Notes: C-Systems C-Window." *Lifelines*, 26 (January 1984).
230 Wilkinson, James B. "YACC is Alive and Well and Running in RSTS." *VAX/RSTS Professional*, 56 – 61 (October 1983).
231 Willman, Bryan M. "C Clearly With Descriptive Operators." *Dr. Dobb's Journal*, 82: 9 (August 1983).
232 Wilson, Dale. "More on Binary Magic Numbers." *Dr. Dobb's Journal*, 70 – 79 (March 1984).
233 Woods, P. L. "Small-C: An Implementor's Notes and a Bug Corrected." *Dr. Dobb's Journal*, 52: 20 – 21, 32, 33 (February 1981).
234 Yates, Jean L. "Unix and the Standardization of Small Computer Systems." *Byte*, 160 – 166 (October 1983).
235 Zachmann, Mark S. "The MWC-86C Compiler." *PC Magazine*, 124 – 129 (March 20, 1984).
236 Zachmann, Mark S. "The Whitesmith's C Native Compiler." *PC Magazine*, 130 – 137 (March 20, 1984).
237 Zintz, Walter. "A Survey of Unix and C Resources." *Byte*, 212 – 213 (August 1983).

Topical Index to C Bibliography

Advanced topics
004, 009, 011, 016, 017, 020, 027, 029, 047, 058, 073, 074, 120, 121, 123, 128, 129, 146, 161, 192, 193, 195, 199, 215, 216, 230
Algorithms
001, 002, 003, 072, 084, 085, 092, 232
BCPL
028, 123
Benchmarks
072, 079, 080, 081, 082, 084, 085, 110
Bibliographies
237
Book reviews
006, 008, 031, 123, 124, 156, 169, 188, 194, 222, 237

C-like languages
C.Plus
076, 077, 078
tiny-C
061, 083, 092, 110, 130
C compiler reviews (by environment)
CP/M
031, 066, 070, 079, 080, 081, 082, 090, 111, 130, 132, 137, 165, 167, 176, 182, 188
CP/M-86
031, 106, 111
MSDOS (PCDOS)
014, 015, 044, 104, 106, 111, 112, 168, 228, 235, 236
C compilers (by name)
Microsoft C
044, 104, 106, 111, 168
LC (Tandy/Radio Shack)
052, 095
Supersoft C
070, 079, 080, 081, 088, 106, 107, 111, 137, 168
Aztec C II
082, 111, 112, 137
Software Toolworks C/80
082, 094, 137
Intellect Associates C88
104, 106, 111, 168
C-Systems
104, 106, 111, 168, 228, 229
Computer Innovations Ci-C86
104, 106, 111, 112, 168, 228
Telecon
079, 080, 104, 111, 137, 168
Lattice
014, 044, 104, 106, 111, 112, 168
DeSmet
079, 080, 081, 104, 106, 111, 137, 168
Digital Research
079, 080, 081, 104, 106, 111, 137, 168
Mark Williams
079, 080, 081, 104, 106, 111, 137, 168, 235
B D Software C
066, 079, 080, 081, 094, 104, 106, 111, 132, 137, 165, 167, 168, 174, 176, 208
Whitesmith's
079, 080, 081, 104, 106, 111, 137, 168, 182, 236
tiny-C
061, 079, 080, 081, 083, 092, 104, 106, 110, 111, 130, 137, 168
C vs. Pascal comparisons
065, 087, 110, 122, 139, 155, 171, 210
Introductory tutorials
013, 015, 031, 054, 079, 080, 081, 083, 091, 116, 117, 122, 125, 126, 140, 152, 186, 188, 211, 228
Language definition
031, 073, 074, 120, 122, 161, 185, 186, 187, 199
Language extensions
004, 005, 011, 016, 017, 027, 029, 030, 128, 129, 146, 207, 209, 215, 216, 230
Language selection criteria
005, 031, 047, 055, 056, 065, 084, 085, 087, 110, 141, 152, 153, 184, 192, 195, 210
Programming hints (generally brief articles or letters)
038, 049, 057, 075, 096, 115, 147, 158, 158a, 170, 171, 173, 191, 204, 205, 206, 231
Programs (include source code)
command shells
097a, 098, 099, 119, 183, 200, 204
concordance generators
213
CP/M interfacing
024, 067, 090, 137, 148
cross assemblers
046
cross reference generators
166, 219, 220
debuggers
010, 016, 017, 027, 145
disk utilities
050, 143, 144, 162, 163
editor (Ed Ream – ED2)
042, 108, 177, 178
editor (Ed Ream – RED)
179, 180
filters
131, 191, 202, 218
Forth-like language
021
help facilities
214
hardware interfacing
009, 024, 025, 047, 055, 056, 058, 105, 141, 189, 193, 212
operating systems
097a, 098, 099, 159, 183
scientific instrumentation
009, 024, 025, 047, 055, 056, 058, 105, 141, 189, 193, 212
Small-C (Cain)
012, 019, 023, 032, 033, 048, 061, 086, 089, 093, 098, 119, 151, 158, 190, 200, 225, 227, 233
Small-C, v. 2 (Hendrix)
034, 097, 100, 101, 102, 103, 197, 198
Small-VOS
097a, 099, 183
telecommunications
045, 047, 063, 175
time/date functions
025
RATFOR
036, 037, 196
Software design and development
005, 020, 083, 122, 125, 126, 142, 150, 184, 195, 207
Software portability
021, 118, 120, 121, 122, 184, 206, 224
Software (non-compilers) reviews
command shells
036, 064, 067, 114, 134, 135, 138, 157, 183, 196, 203
MINCE text editor
041, 043, 059, 060, 133, 154, 164, 170, 221
SCRIBBLE text formatter

136
other software
183
ED editor
042, 108, 177, 178
RED editor
179, 180
Software tools
026, 036, 126, 183, 196, 223
Unix-related
007, 018, 026, 053, 063, 068, 069, 071, 090, 118, 121, 127, 142, 148, 149, 196, 202, 204, 206, 224, 234
Writing tools
039, 226

Public Domain C Software

At the present time a significant amount of public domain software has been written in the C language. Presented below is a brief synopsis of what is currently available. Each disk is a standard 8-inch CP/M disk or approximately 256K of software. Further information concerning ordering, media charges, contents, and so on can be obtained from the addresses preceding the diskette descriptions.

C Users' Group
415 E. Euclid, McPherson, KS 67460, (316) 241-1065.
Also available from Elliam Associates, 24000 Bessemer Street, Woodland Hills, CA 91367, (818) 348-4278.

The C Users' Group provides some of the classic materials in C. The editor (ED2) by Ed Ream is available in both Small-C and BDS C versions. The Small-C compiler is also available.

Five utility diskettes (Utilities I through V) provide programs for text compression, concordances, keyword-in-context concordance, disk editor (direct disk patcher), WP (ROFF-type) text formatter, NRO (enhanced ROFF-type) text formatter, several disk doctoring and patching programs, programs for data compaction and expansion, XSDIR (extended CP/M directory program), entab and detab (insert and remove tabs) programs, telecommunications programs, and a Unix modem system.

A smaller triad of function diskettes (Functions I through III) provide programs for directed input/output facilities (i.e., pipelines), floating point and long integer extensions to BDS C, a trigonometric function library, and scope (simple full-screen text editor).

The group also has available several diskettes of games (Games I and II and an Adventure diskette) that provide a needed respite from "heavy-duty" C programming. The games available include everything from the original (Crowther/Woods) Adventure to Othello, Hunt the Wumpus, etc.

For a much more specialized and smaller clientele, cross-assemblers are available for the 6809 chip (the Radio Shack color computer), the 6800, and the 1802 microprocessor chips.

An extensive set of diskettes (13 volumes) is devoted to the "Software Tools" of Kernighan and Plauger with the primitives translated into C.

Language enthusiasts can find an implementation of a Forth-like language on the PISTOL (Portably Implemented STack-Oriented Language) diskette.

CPMUG
The CP/M Users Group, 1651 Third Avenue, New York, NY 10028.
Also available from Elliam Associates, 24000 Bessemer Street, Woodland Hills, CA 91367, (818) 348-4278; and New York Amateur Computer Club, Inc., P. O. Box 106, Church Street Station, New York, NY 10008.

Three C diskettes are available from the CPMUG. One (volume 48) is a sampler of material in the BDS C language. A second (volume 53) contains the original Crowther and Woods Adventure game, and the last diskette (volume 85) contains a very sophisticated set of text compression programs.

SIG/M Users Group
available from Elliam Associates, 24000 Bessemer Street, Woodland Hills, CA 91367 (818) 348-4278; and New York Amateur Computer Club, Inc., P. O. Box 106, Church Street Station, New York, NY 10008.

Three C diskettes are available from the SIG/M Users Group. One (volume 60) contains a very sophisticated set of text compression programs. A second (volume 114) contains an expanded version of the PISTOL diskette from the C User's Group, and the last diskette (volume 118) contains a C-based implementation of an experimental object-oriented language XLISP.

Some Other Sources of C Software

Below are listed some of the places you may find relatively inexpensive C software, usually including source code.

Edward K. Ream
1850 Summit Avenue
Madison, WI 53705
(608) 231-2952

Ed Ream can supply diskettes for his enhanced RED editor with source code for Small-C, BDS C, Aztec C, and the Digital Research C compilers.

Western Wares
Box C
Norwood, CO 81423
(303) 327-4898

Western Wares provides two diskettes of C utilities including a disk sector editor/dumper program (VIEW diskette) and a set of file utility programs (C-PACK diskette).

Northwest Microsystem Design
P. O. Box 10853
Eugene, OR 97401
(503) 484-7129

A full-screen editor (CSE) is available from Northwest Microsystem Design.

Blaise Computing
2034 Blake Street

Berkeley, CA 94704

An extensive set of tools for the IBM PC is available from Blaise Computing. These tools, available in three packages (C Tools, C Tools II, and View Manager), include such things as strings, screen access, access to BIOS IBM/PC, general DOS interface, and screen support.

Greenleaf Software
2101 Hickory Drive
Carrollton, TX 75006

Over 200 functions for strings, graphics, etc. (IBM PC) are available on the Greenleaf Functions diskette.

Que Corporation
7960 Castleway Drive
Indianapolis, IN 46250

The book *The C Programmer's Library* has available a companion diskette including such topics as recursion, linked lists, sorting algorithms, Huffman text compression, etc.

Dedicated Micro Systems
P. O. Box 481
Chanute, KS 66720
(316) 431-0018

Two diskettes provide extended precision floating point extensions to BDS C. The BDS C compiler is also available.

J. E. Hendrix
Box 8378
University, MS 38677-8378

Mr. Hendrix can provide diskettes for his Small-VOS operating system, for his Small-C compiler, and for a set of software tools written in Small-C (small-tools).

Algorithmic Technology
P. O. Box 278
Exton, PA 19341-0278
(215) 363-7028

Algorithmic Technology publishes a massive directory (40+ pages) of public domain software for CP/M-80, PCDOS, MSDOS, BDS C, and Software Toolworks C/80.

Diskette Conversion Firms

Finally, due to the diversity of disk formats available, I have included below three firms who provide diskette format conversion facilities.

Elliam Associates, 24000 Bessemer Street, Woodland Hills, CA 91367, (818) 348-4278.

Fred Greeb, LogiCom, Inc., P. O. Box 27465, Lakewood, CO 80227, (303) 986-6651.

Mycroft Labs, 2369 North Monroe St., Box 68, Suite B-188, Tallahassee, FL 32303, (904) 385-1141.

[*We understand, however, that Mycroft may be phasing out its conversion activities.—ED*]

DDJ

RESORT

by Donald G. Krantz

One of the things I spend a lot of time doing is writing. I write manuals, a column for the local club newsletter, letters, worksheets, and tests for my classes at night school. Aside from my copy of WordStar, the program I use the most is my public domain spelling checker SPELLM11 by Michael C. Adler.*

While I like to write, I am not what you would call a superlative speller. Until I got SPELLM11, I thought "nessesary" [sic] was just fine. In fact, I almost added it to SPELLM11's user dictionary.

One of the chores associated with any spelling checker is the maintenance of the user dictionary. Because much of what I write is technobabble, the checker flags many of the words that I use as unrecognized. The user dictionary to SPELLM11 must be in alphabetical order.

Being a basically lazy person, I wrote the attached program (see the listing on page 90) to read through a document after spelling corrections, strip the speller's flag (a leading tilde ~), and add correct words unknown to the master dictionary DICT.DIC to the user dictionary SPELL.DIC.

I used the sorting technique from *The C Programming Language* by Kernighan and Ritchie (Prentice-Hall, 1978): the lowly binary tree. This is my favorite sorter, as it is easily implemented in C.

The program takes the input document filename from the command line, opens it for input, and creates a temporary file (with extension .$$$) for output. The input file is scanned for tildes, which flag unrecognized words. When a tilde is found, the word following is converted to upper case and added to a binary tree.

As the input file is scanned, it is fed to the temporary file, minus the tildes, with any high-order bits (used in some editors for formatting purposes) intact. When the input file is exhausted, it is renamed with extension .BAK, and the temporary file takes the original input filename.

The user dictionary SPELL.DIC is then opened, read in, and sorted into the tree. Finally, the tree, sorted into alphabetical order, is written back to the user dictionary.

For those unfamiliar with the binary tree, it is important that the files be read in this order. If a sorted array is loaded into a binary tree, the tree becomes extremely unbalanced, degenerating into a linear linked list; the program bogs down, chasing up and down the tree. For best results, a binary tree should have new nodes presented randomly.

"RESORT is a utility which maintains a user dictionary SPELL.DIC in alpha order."

Donald G. Krantz, 2845 42nd Avenue South, Minneapolis, MN 55406.

** SPELLM11 by Michael C. Adler (public domain) is available as Volume 1 from Avos Blind User's Group, 1485 Energy Park Drive, St. Paul, MN 55108—$5 for 5¼-inch (specify format) and $10 for 8-inch ss/sd CP/M—or from AVOS/TCOG BBS, (612) 646-2848, 300/1200 baud, 24 hours, XMODEM protocol.*

The code is fairly straightforward, as C code goes. It is as close to Unix version 7 compatible as I could make it without actually compiling it on Unix version 7 (my VAX is in the shop). A function-by-function breakdown follows.

main( )

main( ) is concerned mostly with shuffling filenames around. It's one of those run-on functions; for that I apologize. The calls to sort( ), sort2( ), and print( ) do all the program's work.

makebak( )

makebak( ) takes the input filespec from the command line and examines it to see if an extension is part of the filespec. If so, the extension is replaced by .$$$ for the temporary filespec and .BAK for the backup filespec. If no extension is included in the original filespec, the appropriate extensions are appended to the backup and temporary filespecs.

sort( )

sort( ) scans the input stream for tildes. When it finds one, it accumulates subsequent alphabetic characters into the array w[ ]. The words then are passed to treeload( ), which assembles the binary tree.

sort2( )

sort2( ) scans the user dictionary using basically the same tokenizing routine as sort( ), except that it accumulates all of the words from the input stream, not just words flagged by tildes. The words again are passed to treeload( ). This routine is the speed bottleneck in the program. Because the user dictionary is already in order, the more words added to the tree from the input document (presumably in random order, which tends to start the tree out in a balanced condition), the faster this phase will run.

treeload( )

This is patterned directly after Kernighan and Ritchie (K & R). (My philosophy of life is that no matter how many times you invent a wheel, a good one will still be round.) This function chases down the binary tree until it finds either a duplicate word or the end of the tree. Should it find the end of the tree, it creates a new node.

print( )

print( ) writes the new user dictionary back to disk. This routine illustrates why I like binary trees so much: you can write such elegant code. Again, it's from K & R. print( ) writes all daughter nodes lower than a given node, then the node itself, and then all daughter nodes higher than itself. Try to do this in BASIC sometime.

getcc( )

getcc( ) transfers one character from the input file to the temporary file and returns the same character to the tokenizer. Note that ints are used to allow ERROR to be passed back to the caller. If chars were used, ERROR (usually −1, or 0xFFFF) would be converted to DELETE (0xFF).

Ian Ashdown (a referee for *Dr. Dobb's*) suggested that the use of the function sort2( ) is a misapplication of the binary tree because of the previously noted tendency towards unbalancing the tree by adding words in order. He favors using a routine (his pseudocode appears in the figure below) to merge the presumably alphabetic word stream from the user dictionary file with an alphabetic stream taken from the binary tree. This method certainly would eliminate the speed bottleneck noted in sort2( ), but it has one drawback: if the user accidentally gets the dictionary file out of order by editing it (it is, after all, an ASCII file), this merge routine will never again produce a sorted output. I opted to trade some speed losses for reliability.

The program shell can be modified easily to do other tasks unrelated to spelling. For example, it would be a matter of adding a few lines of code to do word frequency counts. I have made a few changes to produce an index generator. In fact, I have written a tokenizer for a preprocessor that uses about half the code in this program. **DDJ**

(Listing begins on page 90)

```
/*  ------------------------------------------------------------
                merge( ) – merges two alphabetical streams.
------------------------------------------------------------ */

merge( )
{
    get( a word "word_sd" from SPELL.DIC )
    get( a word "word_bt" from binary tree )
    do
    {
        if( "word_sd" < = "word_bt" ) /* compare words */
        {
            put( "word_sd" in output stream )
            get( another word "word_sd" from SPELL.DIC )
        }
        else
        {
            put( "word_bt" in output stream )
            get( another word "word_bt" from binary tree )
        }
    }
    write(any remaining words from SPELL.DIC or binary tree)
    /* only one of the two can have leftover words */
```

Figure
Ian Ashdown's Pseudocode for Merge

Resort Listing (Text begins on page 88)

```
/*-------------------------------------------------------------
RESORT is a utility which maintains a user dictionary SPELL.DIC
in alpha order. This works in conjunction with the utility
SPELLM11.COM, by Michael C. Adler, as modified for the Osborne
User Groups.

RESORT is Copyright (C) 1984 by Donald G. Krantz. No commercial
use may be made of this program without permission of Author.

Limited license is granted for non-commercial distribution of
this program in either source or object form.
-------------------------------------------------------------
-------------------------------------------------------------*/

#define VERSION "RESORT Version 1.1 (C) 1984 Donald G. Krantz"
#include "stdio.h"

/*-------------------------------------------------------------
stucture node is the basic data type used to describe the
sorted array of words.
-------------------------------------------------------------*/

struct node {
        struct node *lower;
        struct node *higher;
        char *word;
            };

struct node *root;      /* root is the root of the binary tree  */

FILE *inptr;            /* file descriptor to input files       */
FILE *outptr;           /* file descriptor to output file       */

main( argc, argv )
   int argc;
   char *argv[];
{
   char tempfile[ 20 ];         /* temporary text filespec      */
   char bakfile[ 20 ];          /* .BAK filespec                */

        printf("\n\s\n", VERSION );
/* check command line argument */
        if( argc != 2 )         /* one argument only...         */
        {
                printf("\nERROR - Use:\nRESORT d:filename.typ\n\n");
                exit( 1 );
        }
/* open input file */
        if( (inptr = fopen( argv[1], "r", 512 )) == NULL )
        {
                printf("\nERROR - Can't open file \s\n",argv[1]);
                exit( 1 );
        }
/* make .BAK and .$$$ file spec and open temps */
        makebak( argv[ 1 ], bakfile, tempfile );
        unlink( tempfile );                     /* dump old temps */
        if( (outptr = fopen( tempfile, "w" ) ) == NULL )
        {
                printf( "\nERROR - Can't open temporary file\n" );
                exit( 1 );
        }

/* sort input file and strip tildes, transfer to temp file */
       root = sort( NULL );
/* change temp filename to original, original to .BAK */
       fclose( inptr );                         /* close input    */
       if( fclose( outptr ) == ERROR)           /* close temp     */
       {
```

(Continued on next page)

Resort Listing (Listing Continued, text begins on page 88)

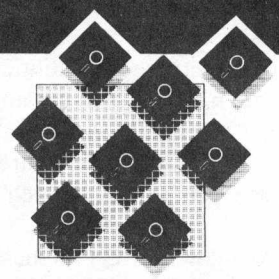
```
                printf( "\nERROR - Disk full - aborting\n" );
                unlink( tempfile );
                exit( 1 );
        }
        unlink( bakfile );                      /* dump old .BAK  */
/* rename: rename( new_file_spec, old_file_spec ) */
        rename( bakfile, argv[1] );             /* make new .BAK  */
        rename( argv[1], tempfile );            /* make new text  */
        if( root == NULL )                      /* no new words   */
                exit( 0 );                      /* so quit        */
/* sort .DIC file into binary tree */
        if( (inptr = fopen( "SPELL.DIC", "r" )) == NULL )
        {
                printf( "\nERROR - Can't open dictionary file\n" );
                exit( 1 );
        }
        root = sort2( root );                   /* sort .DIC      */
        fclose( inptr );                        /* dump .DIC      */
                /* re-write .DIC file   */
        if( (inptr = fopen( "SPELL.DIC", "w", 512 )) == NULL )
        {
                printf( "\nERROR - Can't write to dictionary file");
                exit( 1 );
        }
/* write sorted tree to .DIC file */
        print( root );
        if( fclose( inptr ) == ERROR )
        {
                printf("\nERROR - Disk full - Recheck SPELL.DIC\n");
                exit( 1 );
        }
}

/*---------------------------------------------------------------
makebak() creates filespecs for a temporary file and a backup file
to be used in filtering applications like RESORT. It's kind of
a generic function.
---------------------------------------------------------------*/

makebak( orgfile, bakfile, tempfile )
   char *orgfile, *bakfile, *tempfile;
{
        strcpy( tempfile, orgfile );
        strcpy( bakfile, orgfile );
        if( index( tempfile, '.' )  == NULL )   /*.TYP spec'ed? */
        {                                       /* no           */
                strcat( tempfile, ".$$$" );
                strcat( bakfile, ".BAK" );
        }
        else                                    /* yes          */
        {
                strcpy( index( tempfile, '.'), ".$$$");
                strcpy( index( bakfile, '.'), ".BAK");
        }
}


/*---------------------------------------------------------------
sort() reads the input file and puts the words with leading
tildes into the binary tree.
---------------------------------------------------------------*/

struct node *
sort( p )
   struct node *p;
{
```

```
        static char w[100];
        register int i;
        static int c;

                while( TRUE )
                {
/* find tilde   */
                while((((c = getcc( inptr )) & 0x7F) != '~') &&
                  (c != ERROR))
                        ;
/* accumulate subsequent word into 'w' */
                i = 0;          /* i = letter count in word */
                while( (i == 0) || (isalpha( c ) ) )
                {
                        c = getcc( inptr );
                        if( c == ERROR )
                                return( p );
                        c = toupper( 0x7F & c );
                        if( isalpha( c ) == FALSE )
                                break;
                        w[ i++ ] = c;
                }
                if( i == 0 )
                        continue;
                w[ i ] = '\0';
/* put word into binary tree */
                p = treeload( w, p );
        }
}


/*-------------------------------------------------------------
sort2() reads the .DIC file and puts the words into the binary
tree.
-------------------------------------------------------------*/

struct node *
sort2( p )
   struct node *p;
{
   static char w[100];
   register int i;
   static int c;

        while( TRUE )
        {
                i = 0;          /* i=letter count in word */
/* loop for alphabetic data, skip non-alpha */
                while((i == 0) || (isalpha( c )))
                {
                        c = getc( inptr );
                        if( c == ERROR )
                                return( p );
                        c = toupper( 0x7F & c );

                        if( isalpha( c ) == FALSE )
                                break;
                        w[ i++ ] = c;
                }
                if( i == 0 )
                        continue;
                w[ i ] = '\0';
/* put word into binary tree */
                p = treeload( w, p );
        }
}


/*-------------------------------------------------------------
treeload( w, p ) loads the tree with words, placing them in
alphabetical order.
-------------------------------------------------------------*/
```

Resort Listing (Listing Continued, text begins on page 88)

```
struct node *
treeload( w, p )
   char *w;
   struct node *p;
{

        if( p == NULL )
/* we are at end of tree, need to add a node for this word */
        {
                if((p = alloc( sizeof( struct node ) )) == NULL)
                {
                        printf("\nERROR - Out of memory");
                        exit( 1 );
                }
                if((p->word = alloc( strlen( w ) + 1 )) == NULL)
                {
                        printf("\nERROR - Out of memory");
                        exit( 1 );
                }
                p->lower = NULL;
                p->higher = NULL;
                strcpy( p->word, w );
                return( p );
        }
        else if( strcmp( p->word, w ) > 0 )
/*  word 'w' is lower than the word at this node, look lower */
                p->lower = treeload( w, p->lower );
        else if( strcmp( p->word, w ) < 0 )
/* word 'w' is higher than the word at this node, look higher */
                p->higher = treeload( w, p->higher );
/* if word 'w' is identical to the word at this node, we do */
/* nothing. In all cases, we return the current node pointer. */
        return( p );
}


/*-----------------------------------------------------------------
print() writes the words in alpha order back into the .DIC
file.
-----------------------------------------------------------------*/

print( p )
   struct node *p;
{
/* print all words lower than current node first */
        if( p->lower != NULL )
                print( p->lower );

/* print the word at the current node */
        fprintf( inptr, "\s\n", p->word );

/* print all words higher than the current node */
        if( p->higher != NULL )
                print( p->higher );
}


/*-----------------------------------------------------------------
getcc() gets chars from the infile, and writes all but tildes
to the outfile. This function does not disturb any high bits
set for parity or formatting.
-----------------------------------------------------------------*/

getcc( fd )
   int *fd;
{
   int c;
        c = getc( fd );
        if(((c & 0x7F) != '~') && (c != ERROR))
                fputc( c, outptr );
        return( c );
}
```

End Listing



Windows for C, Version 2.00

Company: Creative Solutions, 21 Elm Avenue, Richford, VT 05476

Computer: IBM PC and MSDOS (various C compilers)

Price: $150.00, demo disk and manual: $30.00

Reviewed by Ian Ashdown

We have good news: If you are running MSDOS on your IBM PC and programming C, you probably will be pleased to learn that Windows for C can offer you a convenient and reliable means of implementing windows for your text-handling programs.

Windows are, of course, those rectangles on a terminal's screen that enclose information not directly related to other information shown. In essence, they are screens in their own right, showing anything from a different part of the same text file to the operation of a completely independent, concurrently running program. Properly implemented, they can greatly enhance the utility of everything from menus and help facilities to full-screen editors, spreadsheets, data base managers, and multitasking operating systems.

As such, it behooves the C programmer to add windowing to his/her library of C functions. Windows for C greatly eases this effort by providing over 30 such functions for the IBM PC with MSDOS. Using this software, you can define and manipulate windows for text applications, treating each window as a separate screen and having the number of screens limited only by the available memory. If you don't need to display a particular window at any time, simply erase it from the screen. Because it is stored in memory, complete with current contents, it can be retrieved later and displayed.

In defining a window, you can specify its location and size and also define the visual appearance of its border and background. These characteristics are not static—you can dynamically redefine them at any time during program operation. Windows can also be overlapped, nested, cleared, and removed from the screen as required, all without affecting any other nonoverlapping windows.

The contents of the windows are whatever textual information you want; Windows for C supports all of the text mode capabilities of the IBM monochrome and graphics display adapters. Video attributes such as intensity, underline, reverse, blink, and color can be controlled on a character-by-character basis. The text can be written to the window with or without word wrap, and both vertical and horizontal scrolling are supported. If desired, you can also have automatic scrolling of text in a full window when the cursor reaches the window's border.

Each window has its own virtual cursor, with the cursor position always kept in memory. Accessing the window automatically moves the real screen cursor to the position of the virtual cursor, and any manipulation of the screen cursor in the window automatically updates the position of the virtual cursor in memory.

Text can come from any source—the terminal (or other input device) or a disk file. Once displayed in a window, text can be manipulated and, if desired, transferred to other windows. No general-purpose text editing functions are provided with Windows for C, but source code is provided for the routines to read the disk files into memory and into a window. Using this basic code as an example, the well-seasoned C programmer can write the interfaces necessary to link the functions provided by Windows for C to other application programs.

Windows for C displays and manipulates text by defining every window as a data structure, with variables for window position and size, virtual cursor position, background, text margins, word wrap and scroll switches, and so on. Also included in the window structure are pointers to the associated file and border structures. The file structure contains the name of the device or file being accessed and pointers to the part of the file currently displayed in the window, while the border structure defines the characters that form the horizontal and vertical sides and the corners of the window border.

The functions provided by Windows for C were written in both C and assembly language, and so far we have not found any bugs in them. The video output is fast and clean, showing no snow with either the graphics or monochrome display. (Some skewing is evident when the text is scrolled sideways, but this is not objectionable.) The text is sent directly to the video display buffer using line-oriented assembly language subroutines; word wrap, attribute selection, and automatic scrolling are user selectable.

A software package must be judged in large part by its documentation; obscure language, incorrectly described program actions, and undocumented features are the basis of many a programmer's nightmares. Happily, Windows for C is documented in well-written and readable English, and it appears to be all there. The manual we received consisted of 53 double-sided 8½ × 11-inch pages in a three-ring binder. The main topic being the window functions, each one was clearly and concisely documented in the style of Bell Laboratories' *Unix Programmer's Manual*, with entries for the function's name, usage, operation, definition, what it returns, and cautionary notes. For a reference manual, this style of

writing is ideal.

Moreover, in addition to the perfunctory "Getting Started," "Tutorial," and "Advanced Capabilities" sections, the manual has an appendix entitled "Mnemonic Abbreviations." This is a most welcome feature. How often have you gone in search of the meaning of an abbreviation such as "csr"? Windows for C lists them all in one place ("csr" refers to the screen cursor). Many other software documentation writers, take note!

To summarize the good news, we liked Windows for C: it is a very nice software package that has obviously been well thought out and very cleanly executed.

. . . and with the good news, a bit of the not-so-good. As previously noted, Windows for C was written for the IBM PC. It can also be used with unspecified "video-compatible" computers running MSDOS. However, since the manual does not say what IBM work-alikes are video compatible, you would do well to enquire before purchasing the package if you plan on using anything other than an IBM PC.

Most of the functions included in Windows for C are in object code form. Since each C compiler uses a different library format, Creative Solutions is obliged to distribute a different version of Windows for C for each one. C compilers currently supported are those distributed by Lattice, Microsoft, Computer Innovations (C86), and DeSmet.

You might also bear in mind that windows are very much in demand for personal computer software these days. Considering Microsoft's Windows, Digital Research's windowing version of Concurrent CP/M, and the host of similar software packages soon to follow, we must ask whether a software package that provides windowing functions for text modes only is desirable or sufficient. Mind you, these others are really operating system environments, whereas Windows for C enables you to develop stand-alone software to run under MSDOS.

The above notwithstanding, we can say that Windows for C is probably a worthwhile investment for the serious C programmer. Under the terms of Creative Solution's program license agreement, the functions provided can be linked into the programmer's own software efforts and marketed without royalties or Creative Solution's copyright notice. As such, Windows for C is a professionally oriented set of programming tools and one that we can recommend.

Special thanks is extended to Robert Koorbatoff, free-lance programmer, who assisted in this review by providing his time, expertise, and Microsoft C compiler.

MYCHESS, MSDOS Version 1.0

Company: The Software Toolworks, 15233 Ventura Blvd., Suite 1118, Sherman Oaks, CA 91403

Computer: IBM PC, PC XT, or PCjr

Price: $34.95

Reviewed by Ronald G. Parsons

MYCHESS is a chess playing program for the IBM PC. Other versions are also available for CP/M. On the IBM PC with a graphics card, the display shows very attractive representations of the various chess pieces in high-resolution black and white. The system is configurable to use medium-resolution graphics or an ASCII representation on a monochrome display, but the attractiveness and readability are much degraded. The authors claim a U.S. Chess Federation rating of 1568.

You can set the skill level of the program by altering the number of half moves (ply depth) that the program looks ahead. It can play black or white, from the beginning or from a setup position, or you can interrupt a game, save it, and resume it later. The program can even play itself. The moves are recorded on the screen (showing the last eight moves), or the moves can be recorded on a printer or listed on a file at the end of the game for later viewing.

You start the game by answering a serious of questions regarding the game setup. Then the chess board is displayed, and "white" plays. The notation identifies each square by a file lable (column) A – H and a row label 1 – 8; for example, the white king starts on E1. Each move is denoted by a starting square and a destination; e.g., white king's pawn to king four would be E2 – E4. Illegal moves are rejected with a beep. Pawns can be promoted, but only one queen of each color may exist at any time.

The program is easy to play and learn. The moved piece flashes several times in its new position so that an eye blink does not cause you to miss a move. The move is also recorded on the right of the screen. I would like, however, to see several features added to the program. There is no way to tell whether you forgot to hit the enter key or the computer is thinking—a visible cursor would be helpful. There is no list of pieces captured—a command to list the captured pieces would be helpful. An on-line help feature to refresh your memory of the available commands also would be nice. A mouse interface would at least be cute.

The major impression of the type of game played by MYCHESS is one of attrition. In fact, the program looks at capture moves first, which reduces the number of positions it must analyze to determine the next move. The priorities of move generation used by MYCHESS verifies this. The top four priorities of possible moves are:

(1) Best variation from a previous iteration
(2) Winning or even captures (thus the attrition)
(3) Castle moves
(4) *En passant* captures

In addition, a one-ply search is followed by two-ply searches, and so on, so that if a move is forced, the best move of the previous iteration is available. Also each successive iteration is based on the best variation predicted by the previous iteration. The program examines one extra ply before backing up from a best variation if the side to move has anything to capture.

I noted only one bug and it was minor. One start-up option is to list the best variations for a selection of moves. If the number of moves suggested is more than three or so, the suggested moves wrap around and cover the board part of the screen.

MYCHESS is easy to use and presents a pleasant interface to the user. Without a graphics board and display, the program is much more difficult to use. I would recommend it especially to beginning chess players.

DDJ

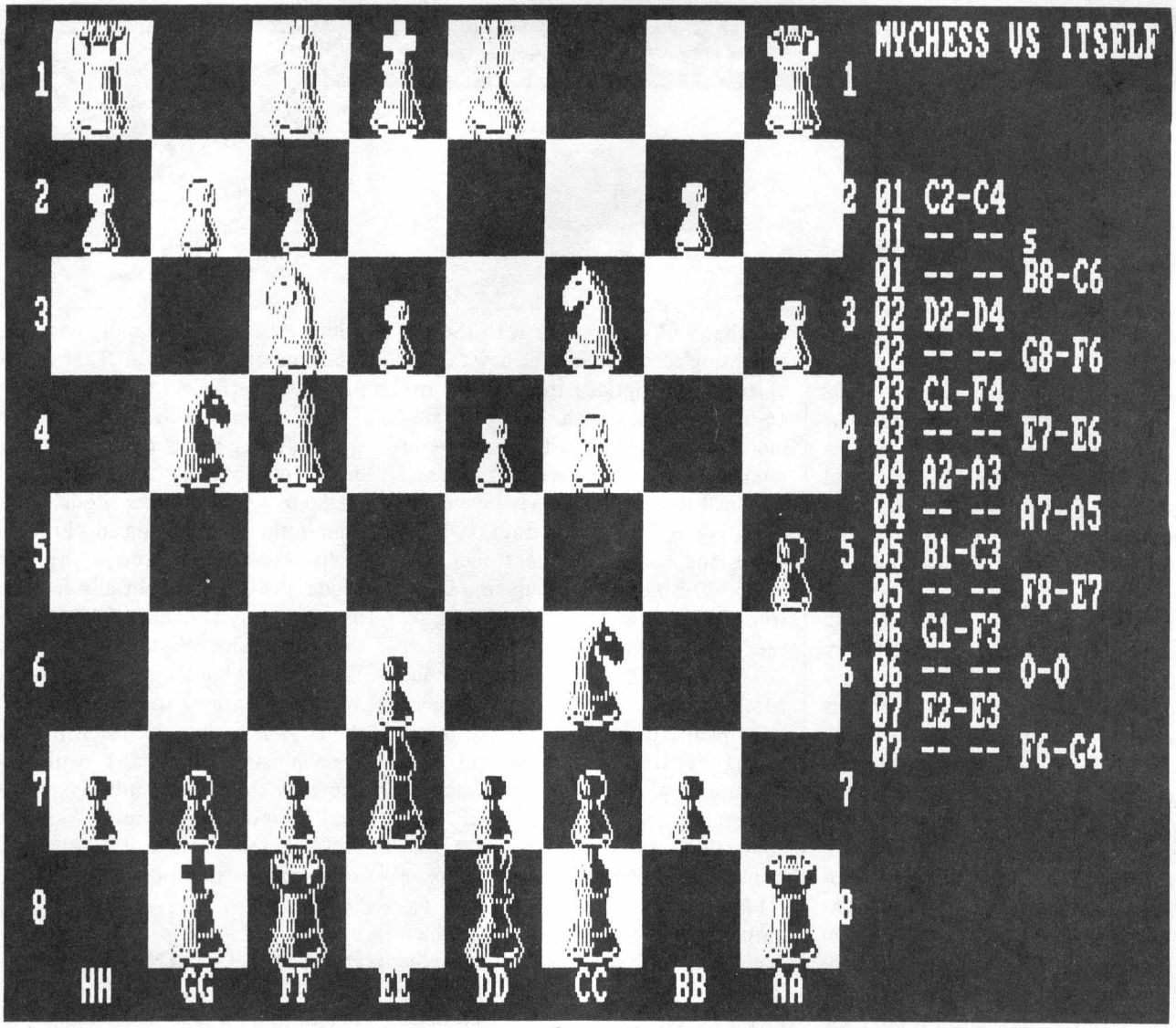
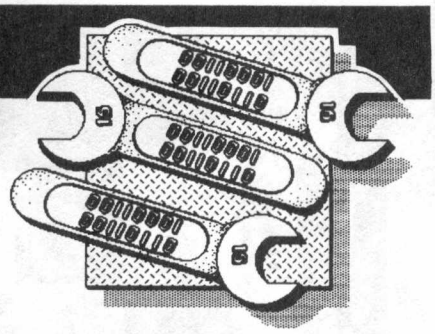


Figure 1
MYCHESS vs Itself

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

The IBM PC/AT

IBM has confounded its critics and finally lowered the boom on "PC-compatible" manufacturers by releasing its next generation of high-end personal computers. From the hardware point of view, the PC/AT turned out to be pretty much as expected. It is conservatively designed with a 80286 CPU running at only 6 MHz, a 20-megabyte hard disk, 80-track 1-megabyte floppy disk drives, and 512K of RAM standard. As far as price goes, the PC/AT is clearly a "fighting machine"—only $6000 out the door, just about the cost of a fully equipped and much slower PC/XT only a few months ago.

Performance-wise, the PC/AT has a very snappy feel. A preliminary computation-bound benchmark (generating the first 1000 primes) that I ran in PC/Forth yielded a time of 24.7 seconds on the "old" PC and 8.9 seconds on the PC/AT. This seems to verify the claims in the trade rags of a threefold speed advantage over the older machine. Real speed freaks will want to plug in an 80287 numeric coprocessor for number crunching; this chip is still pretty scarce and costs about $375 (compared to $175 for the 8087).

Physically, the PC/AT is slightly larger and heavier than the PC/XT. The keyboard has been extensively reorganized. The "Enter" key is now enormous, the "Esc" and "Backslash" keys have been moved into the upper right corner, and the "Caps Lock" key has the little LED indicator that everyone has been crying for. Most important of all, a keyboard lock now allows executives to go out to lunch, leave their PC/AT turned on, and not worry about anyone stealing their Lotus 1–2–3 worksheets.

The degree of upward hardware compatibility on the PC/AT from the previous PC is nothing short of astonishing. The PC/AT has a dual-nature bus that will work either with the old expansion cards or with the new family of expansion options that exploit the 16-bit wide data path. The programmer's view of the hardware-software interface is, for all practical purposes, identical to the old PC. My own company's PC/Forth product, which drives the 6845 video controller and video refresh buffer directly and is extremely hardware dependent, runs on the PC/AT without a hitch.

The PC/AT's Technical Reference Manual is already available and contains plenty of required and/or interesting reading. Systems-level programmers particularly will want to inspect the sections on the dual DMA controllers, the RT/CMOS RAM, and the keyboard controller. The 150-page (!) BIOS contains some instructive examples of 80286 programming, which previously have been hard to come by. The Intel book *iAPX 286 Programmer's Reference Manual* is also helpful here.

The PC/AT is delivered with PCDOS 3.0, which appears to be a sort of warmed-over and expanded version of PCDOS 2.0. Support for partitioning the hard disk has been improved considerably. Also, the FORMAT command has been idiot-proofed, making it less easy to accidentally format the hard disk and erase all your files. PCDOS 3.0 adds a few new DOS commands (ATTRIB and LABEL), a few new CONFIG file commands, some support for file sharing in a networking environment . . . nothing very dramatic. However, IBM clearly has signaled its intention to move the PC/AT into the international market with standard DOS commands to redefine the keyboard and time/date display formats for the major European languages.

DEBUG and LINK don't appear to have changed at all. A RAM-DISK driver is now provided as a standard utility. Also provided, sadly enough, is the same old horrible EDLIN that we first met in PCDOS 1.0.

To ensure upward compatibility with software written for the previous models of IBM PC, PCDOS 3.0 runs the 80286 in "Real Address Mode" rather than in the more advanced "Protected Virtual Address Mode." In Real Mode, the 80286 essentially functions like a slightly enhanced 8086 and has the same 1-megabyte address space. We can only hope that the alternative operating systems soon to appear for the PC/AT, such as Xenix, will exploit the sophisticated support available in Protected Mode for multitasking, virtual memory, and memory protection—thereby wringing all the available computing power out of the 80286.

Periscope PC Debugger

Brett Salter was kind enough to send me an advance copy of his "Periscope" hardware/software debugger for the IBM PC. This is an inexpensive, easy to use, and deceptively simple-looking program analysis tool. Periscope consists of a "short" board containing 16K of protected RAM and some additional logic, a cable with a pushbutton, and a disk containing a debugger/control program. Using Periscope, you can push the button at any time to grab control away from an executing program, after which you may search memory, disassemble programs, set breakpoints, or display memory and registers.

A particularly nice feature of Periscope is the ability to set breakpoints on a specific iteration of a loop, an access to a memory location, the contents of a memory location, or the contents of a register. The program can read the standard symbol tables generated by the PCDOS Linker, allowing the user

to set breakpoints either by name or by location. Dual monitors are supported when present, facilitating the debugging of graphics routines.

The manual is particularly well done, with clear installation instructions, a tutorial, detailed descriptions of all debugger commands, notes on debugging theory and techniques, a summary of all error messages, and a quick reference card. I continue to be astonished at how far the general level of documentation quality has risen since I built my first Imsai 8080 microcomputer. Periscope is available for $295 from Data Base Decisions, 14 Bonnie Lane, Atlanta, GA 30328 (404) 256-3860.

Grandson of Floating-Point Benchmark

I thought I was rid of the Savage Floating Point Benchmark subject forever after the publication of the expanded table of results in the August 1984 *DDJ*. However, that table only incited *DDJ*'s loyal readers to send in even more timings. I guess I'll keep collecting for yet another while and publish the table again in early 1985. We're still looking for a COBOL listing, guys! Latest impressive mainframe result: 0.016 seconds with error 5E-7, Unix Basic Compiler, double precision, VM/UTS operating system, Amdahl 5860 CPU.

Counting Cycles

Bob Smith of Qualitas Inc. wrote in to chide me for a foolish statement I made in another column. I noted that, according to the Intel documentation, the sequence

```
SHL     BX,1
SHL     BX,1
SHL     BX,1
SHL     BX,1
```

occupied 8 bytes and required 8 cycles to execute, while the sequence

```
MOV     CL,4
SHL     BX,CL
```

which looks more elegant on paper, occupied only 4 bytes but required 28 cycles. Mr. Smith comments:

"The reference to instruction timings . . . prompts me to write about this counter-intuitive topic. It's tempting to rely solely upon the published raw instruction timings; however, the picture is more complicated. Additional effects, such as draining and refilling the 8088's four-byte pre-fetch instruction queue, can play an overriding role.

"As it turns out, the two-instruction combination MOV CL,4 . . . SHL BX,CL is actually faster than four SHL BX,1 instructions, although the raw instruction timings suggest the opposite. The reason appears to be that, for the four SHL BX,1 instructions, their small size (2 bytes) and short execution time (2 clocks) drain the queue quickly—so quickly, in fact, that their execution time plus the time spent refilling the queue exceeds the time for the two-instruction version.

"These timings can be measured directly with a high-resolution software timer as described in our article 'Life in the Fast Lane,' *PC Tech Journal*, Vol. 1, No. 7 (April 1984). Please note that in the article the code segments in the listings of Figures 4 and 5 were inadvertantly switched.

MSDOS Programming Pearl of the Month

When writing applications for MSDOS 2.0, many programmers have been disappointed in the display speeds available through the standard MSDOS output calls and have resorted to direct control of the hardware for better performance. It turns out that you can obtain substantially increased display speeds for "well-behaved" application programs on many MSDOS 2.0 systems simply by setting the "raw output mode" bit in the driver's device information word. This bit tells MSDOS not to check for a CTRL-C from the keyboard between each character it transfers to the output device. Listing One (page 100) provides the necessary code to set the raw output mode bit. Thanks to Dave Angel of Wang for this tip.

DOS 2.0 Filters

As a followup to August's "Clean" filter, this month we are printing the "TK" tokenizing filter contributed by Jim Mott of Sunnyvale, California (Listing Two, page 100). A detailed description of the operation of the filter with examples is contained in the listing itself. Jim writes:

"One warning: If you use the TK filter after the FIND filter with too little space on the %PIPE## disk, it will fail without any error messages. FIND doesn't seem to check the number of bytes written compared to the number requested. If the disk is full, FIND just stops writing to its pipe. It doesn't even tell you that something is wrong. When TK starts, the input file is only half there, so it doesn't have all the data to work with. If anyone would like to get a copy of the TK program on disk, tell them to send along $15 to cover disk, postage, and handling." DDJ

16-Bit (Text begins on page 98)

Listing One

Selecting raw output mode for standard output under MSDOS 2.0 disables DOS's check for CTRL-C pending on the keyboard between each character transmitted to the standard output, thereby speeding up displays.

```
;
; Select Raw Output Mode on Standard Output Handle
;
mov     bx,1            ; i/o control read for
mov     ax,4400h        ; "device information"
int     21h
mov     dh,0            ; set upper byte of DH = 0
or      dl,20h          ; set raw mode bit in DL
mov     bx,1            ; i/o control write of
mov     ax,4401h        ; "device information"
int     21h
```

End Listing One

16-Bit

Listing Two

```
                        PAGE    55,132
                        TITLE   TK - Token Parsing Filter

                ;
                ;   TK --- A Simple Token Parsing Filter for DOS 2.0
                ;
                ;   (C) Copyright 1984 by Jim Mott
                ;                       3710 Slopeview Drive
                ;                       Sunnyvale, CA. 95148
                ;                       (408) 274-2620
                ;   All rights reserved.  Permission granted to use this software for
                ;   personal, noncommercial purposes only.
                ;
                ;
                ;   This program is designed to be a filter for DOS 2.0.
                ;   It will tokenize its input and allow subsetting and/or
                ;   single token per line output.
                ;
                ;   The format of the command is:
                ;
                ;   TK {/RJx | /LJx} {/0} {{/v} | {/v/v}}
                ;     where /RJx means right justify all tokens to x positions
                ;           /LJx means left justify all tokens to x positions
                ;                In the above two entries x must be in [1..15]
                ;           /0   means output one token per line
                ;           /v   means select token v for output.  You may select any number,
                ;                up to 255, of tokens to output.  Repeats are allowed and you
                ;                may change the order of the input tokens on the output line.
                ;
                ; For example, to extract a list of users from a VM directory file and write
                ;     a sorted list of them without passwords to the printer the following
                ;     command land line would be used.
                ;
                ; FIND "USER " < DIRECT.VM | TK/LJ8/2/4/5/6/7/8/9 | SORT > PRN
                ;
                ;
                ; For example, to find a list of all sub directories of the current directory
                ;     sorted by sub directory name we could use the following command line.
                ;
```

```
                                ; DIR | FIND "<DIR>" | TK/LJ8/1/3/4 | SORT | MORE
                                ;
                                ;
                                ; For example, to generate a sorted list of all words used in a document with
                                ;     one word per line we could use the following line.
                                ;
                                ; TK/RJ8/0 < FOOBAR.DOC | SORT | MORE
                                ;
                                ;
                                ; A note on comments and spelling.  I am a very poor speller so I
                                ; take the view that programs can either well documented with
                                ; creatively spelled words or not documented with perfectly spelled words.
                                ; I choose the creative approach and am satisfied when the opcodes are
                                ; spelled correctly.
                                ;
                                ;
0000                            STACK   SEGMENT PARA    STACK   'STACK'
0000    08 [                            DB      8 DUP('Jim Mott  (408) 274-2620')
            4A 69 6D 20
            4D 6F 74 74
            20 20 28 34
            30 38 29 20
            32 37 34 2D
            32 36 32 30
                    ]
00C0                            STACK   ENDS
                                ;
                                ;
0000                            DSECT   SEGMENT PARA    'DATA'
0000    FF [                    BUFFER  DB      255 DUP('?')    ;WHERE TO PUT THE DATA
            3F
                ]
00FF  20                                DB      ' '             ;BE SURE TO END SCAN CORRECTLY
                                ;
0100  0000                      GLEN    DW      0               ;LENGTH OF GBUFF
0102  0104 R                   GBPTR   DW      GBUFF           ;POINT TO START OF BUFFER
0104    FF [                    GBUFF   DB      255 DUP('G')    ;BUFFER USED BY BUFGET
            47
                ]
                                ;
0203  ??                        FLAG1   DB      ?               ;OPTIONS ENABLED
= 0001                          F1RJ    EQU     01H                     ;RIGHT JUSTIFY TOKENS
= 0002                          F1LJ    EQU     02H                     ;LEFT JUSTIFY TOKENS
= 0004                          F1ONE   EQU     04H                     ;OUTPUT ONE TOKEN PER LINE
= 0008                          F1SUB   EQU     08H                     ;SUBSTRING FUNCTION REQUESTED
= 0010                          F1WORK  EQU     10H                     ;FILL TRAILING SPACES
= 0020                          F1OERR  EQU     20H                     ;ERROR IN OPTIONS STRING
= 0040                          F1EOF   EQU     40H                     ;END OF FILE ON STANDARD INPUT DEVICE
= 0080                          F1QEOF  EQU     80H                     ;QUE THE END OF FILE
                                ;
0204  ??                        SPACES  DB      ?               ;NUMBER OF TRAILING SPACES REQUIRED
                                ;
0205  ??                       TOKSIZ  DB      ?               ;TOKEN SIZE IF (F1RJ OR F1LJ)
                                ;
0206  03                        THREE   DB      3               ;LENGTH OF EACH ENTRY
0207  ??                        TOKCNT  DB      ?               ;COUNT OF TOKENS IN TABLE
0208  02FD [                    TOKTBL  DB      3*255 DUP('0')  ;TABLE OF TOKEN POINTERS AND LENGTHS
            30
                ]
                                ;
0505  ??                        OUTCNT  DB      ?               ;NUMBER OF SUBSETTING ENTRIES IN OUTERS
0506    FF [                    OUTERS  DB      255 DUP('1')    ;LIST OF TOKEN NUMBERS TO OUTPUT
            31
```

(Continued on next page)

Listing Two

```
                        ]
                                        :
0605  ????                    TOKPTR  DW      ?               ;POINTER TO FREE TOKEN SPACE
0607  0384 [                 TOKENS  DB      900 DUP('2')    ;STRING SPACE OF TOKENS
                  32
                        ]
                                        ;
098B  54 4B 3A 20 49 6E      MSGVER  DB      'TK: Incorrect DOS version.  Must be at least 2.00.'
      63 6F 72 72 65 63
      74 20 44 4F 53 20
      76 65 72 73 69 6F
      6E 2E 20 20 4D 75
      73 74 20 62 65 20
      61 74 20 6C 65 61
      73 74 20 32 2E 30
      30 2E
09BD  0D 0A 24                          DB      0DH,0AH,'$'
09C0  54 4B 3A 20 49 6E      OPTMSG  DB      'TK: Incorrect parameters given.'
      63 6F 72 72 65 63
      74 20 70 61 72 61
      6D 65 74 65 72 73
      20 67 69 76 65 6E
      2E
09DF  0D 0A                             DB      0DH,0AH
= 0021                       OPTMGL  EQU     $-OPTMSG
09E1  54 4B 3A 20 4E 6F      NOROOM  DB      'TK: No room for user on device.'
      20 72 6F 6F 6D 20
      66 6F 72 20 75 73
      65 72 20 6F 6E 20
      64 65 76 69 63 65
      2E
0A00  0D 0A                             DB      0DH,0AH
= 0021                       LNOROOM EQU     $-NOROOM
0A02  20                     CHRSPA  DB      ' '             ;A SPACE TO OUTPUT
0A03  0D 0A                  CHRCLF  DB      0DH,0AH         ;<cr><lf> SEQUENCE
0A05                         DSECT   ENDS
                                        ;
                                        ;
0000                         CSECT   SEGMENT PARA    'CODE'
                                        ASSUME  CS:CSECT,DS:DSECT,SS:STACK
0000                         MAIN    PROC    FAR
0000  1E                               PUSH    DS              ;SET UP A RETURN ADDRESS
0001  2B C0                            SUB     AX,AX           ;WE WANT TO RETURN TO DS:0000
0003  50                               PUSH    AX
0004  B8  ---- R                       MOV     AX,DSECT        ;POINT TO START OF DATA AREA
0007  8E D8                            MOV     DS,AX           ;MAKE ASSUME AND REALITY AGREE
0009  B4 30                            MOV     AH,30H          ;GET DOS VERSION NUMBER
000B  CD 21                            INT     21H             ;CALL OS TO GET IT
000D  3C 02                            CMP     AL,2            ;IS IT AT LEAST 2.00
000F  7D 09                            JNL     MAIN00          ;YES - GOOD ENOUGH
0011  8D 16 098B R                     LEA     DX,MSGVER       ; NO - POINT TO THE BAD DOS VERSION MESSAGE
0015  B4 09                            MOV     AH,9            ;      DOS 1.?? FUNCTION TO PRINT AN ERROR
0017  CD 21                            INT     21H             ;CALL THE OPERATING SYSTEM
0019  CB                               RET                     ;AND DO A LONG RETURN
001A  E8 006A R              MAIN00: CALL    OPTIONS         ;PARSE THE OPTIONS (AT ES:80) AND SET FLAGS
001D  8C D8                            MOV     AX,DS           ;MAKE ES AND DS THE SAME NOW
001F  8E C0                            MOV     ES,AX           ;SO THE STRING MOVES WORK NICELY
0021  F6 06 0203 R 20                  TEST    FLAG1,F1OERR    ;WAS THERE AN ERROR IN THE OPTIONS?
0026  74 10                            JZ      MAIN01          ; NO - THEN GO WITH THIS BABY
0028  8D 16 09C0 R                     LEA     DX,OPTMSG       ;YES - POINT TO OPTIONS ERROR MESSAGE
002C  B9 0021                          MOV     CX,OPTMGL       ;      GET LENGTH OF MESSAGE
002F  BB 0002                          MOV     BX,2            ;      ERROR OUTPUT DEVICE HANDLE
0032  B4 40                            MOV     AH,40H          ;      OPERATING SYSTEM FUNCTION NUMBER
0034  CD 21                            INT     21H             ;      CALL THE OPERATING SYSTEM
0036  EB 29                            JMP     SHORT MAIN03    ;      AND RETURN AS DONE
```

```
0038  C6 06 0207 R 00       MAIN01: MOV     TOKCNT,0        ;WE DON'T HAVE ANY TOKENS IN THE TABLE
003D  8D 06 0607 R                    LEA     AX,TOKENS       ;POINT TO START OF TOKEN WORK AREA
0041  A3 0605 R                      MOV     TOKPTR,AX       ;SAVE POINTER TO NEXT FREE BYTE IN TOKEN SPACE
0044  E8 027F R                      CALL    BUFGET          ;READ IN A BUFFER
0047  F6 06 0203 R 40                TEST    FLAG1,F1EOF     ;IS THERE ANY DATA IN THE READ BUFFER
004C  75 13                          JNZ     MAIN03          ; NO - WE ARE DONE WITH THIS PROGRAM THEN
004E  49                             DEC     CX              ;YES - IGNORE THE TRAILING <cr>
004F  7E E7                          JLE     MAIN01          ;IF LENGTH IS ZERO OR WORSE JUST GET NEXT LINE
0051  8D 1E 0000 R                   LEA     BX,BUFFER       ;POINT TO THE FIRST BYTE OF THE DATA
0055  E8 0103 R             MAIN02: CALL    NEXTOK          ;GET THE NEXT TOKEN
0058  0B C9                          OR      CX,CX           ;ARE WE DONE WITH THIS LINE YET?
005A  75 F9                          JNZ     MAIN02          ; NO - GET YET ANOTHER TOKEN
005C  E8 01A2 R                      CALL    WRITE           ;YES - WRITE THE LINE(S)
005F  EB D7                          JMP     SHORT MAIN01    ;AND LOOP FOR THE NEXT LINE
0061  E8 0213 R             MAIN03: CALL    CRLF            ;WRITE A FINAL <cr><lf> SEQUENCE
0064  B0 00                          MOV     AL,0            ;THIS IS THE RETURN CODE TO POST
0066  B4 4C                          MOV     AH,4CH          ;TERMINATE A PROCESS CODE
0068  CD 21                          INT     21H             ;END THIS PROGRAM
006A                         MAIN    ENDP
                             ;
                             ;
                             ; OPTIONS - This subroutine will parse the options passed to the program and
                             ;           set the required bits in FLAG1.  No registers are preserved since
                             ;           we are called only once, before the program has really started.
006A                         OPTIONS PROC    NEAR
006A  C6 06 0505 R 00                MOV     OUTCNT,0        ;INITIALIZE OUTERS COUNT
006F  BE 0081                        MOV     SI,81H          ;POINT TO THE FIRST CHARACTER IN PARMS
0072  26: 8A 04              OPT01:  MOV     AL,BYTE PTR ES:0[SI] ;GET A BYTE FROM PARM STRING
0075  46                             INC     SI              ;POINT TO NEXT BYTE
0076  3C 0D                          CMP     AL,0DH          ;IS IT THE END OF THE STRING?
0078  75 01                          JNE     OPT02           ; NO - GOODY, MORE DATA TO PROCESS
007A  C3                             RET                     ;YES - RETURN TO CALLER THEN
007B  3C 20                  OPT02:  CMP     AL,' '          ;ALLOW SPACES ANYWHERE BEFORE SLASHES
007D  74 F3                          JE      OPT01           ;IGNORE THEN THOUGH
007F  3C 2F                          CMP     AL,'/'          ;WE HAVE TO START WITH A SLASH NOW
0081  74 06                          JE      OPT04           ;IF IT IS A SLASH THEN PROCESS IT
0083  80 0E 0203 R 20        OPTERR: OR      FLAG1,F1OERR    ;OTHERWISE SET THE OPTIONS ERROR FLAG
0088  C3                             RET                     ;AND RETURN
0089  26: 8A 04              OPT04:  MOV     AL,BYTE PTR ES:0[SI] ;GET THE NEXT CHARACTER AFTER SLASH
008C  46                             INC     SI
008D  3C 61                          CMP     AL,'a'          ;IS IT LOWER CASE OR FUNNY?
008F  7C 02                          JL      OPT4A           ; NO - PROCESS NORMALLY THEN
0091  2C 20                          SUB     AL,'a'-'A'      ;YES - MAP LOWER CASE TO UPPER
0093  3C 4C                  OPT4A:  CMP     AL,'L'          ;MIGHT IT BE LEFT JUSTIFY OR NUMERIC
0095  7C 42                          JL      OPTNUM          ;PERHAPS NUMERIC - CHECK IT OUT
0097  75 0E                          JNE     OPT05           ;IT IS NOT LJ FOR SURE
0099  80 0E 0203 R 02                OR      FLAG1,F1LJ      ;ASSUME IT IS LJ FOR THE MOMENT
009E  F6 06 0203 R 01                TEST    FLAG1,F1RJ      ;MAKE SURE THIS ISN'T A DUPLICATE
00A3  75 DE                          JNZ     OPTERR          ;IF RJ ALREADY THEN BIG PROBLEMS
00A5  EB 10                          JMP     SHORT OPT06     ;AND REJOIN COMMON JUSTIFY CODE
00A7  3C 52                  OPT05:  CMP     AL,'R'          ;MIGHT IT BE RIGHT JUSTIFY (RJ)?
00A9  75 D8                          JNE     OPTERR          ; NO - THEN IT IS AN ERROR
00AB  80 0E 0203 R 01                OR      FLAG1,F1RJ      ;YES - ASSUME FOR THE MOMENT IT IS
00B0  F6 06 0203 R 02                TEST    FLAG1,F1LJ      ;MAKE SURE WE AREN'T TRYING TO LEFT JUSTIFY TOO
00B5  75 CC                          JNZ     OPTERR          ;IF WE ARE, THEN WE ARE IN DEEP S..T
00B7  26: 8A 04              OPT06:  MOV     AL,BYTE PTR ES:0[SI] ;GET THE NEXT CHARACTER
00BA  46                             INC     SI
00BB  3C 4A                          CMP     AL,'J'          ;IS IT THE J WE EXPECT?
00BD  74 04                          JE      OPT6A           ;YES - PROCESS IT NORMALLY THEN
00BF  3C 6A                          CMP     AL,'j'          ;IS IT A LOWER CASE J?
00C1  75 C0                          JNE     OPTERR          ; NO - THAT'S TO BAD.
00C3  26: 8A 04              OPT6A:  MOV     AL,BYTE PTR ES:0[SI] ;GET THE FIRST BYTE OF THE NUMBER
00C6  46                             INC     SI
00C7  E8 0247 R                      CALL    DECBIN          ;IS IT A NUMBER?
00CA  72 B7                          JC      OPTERR          ; NO - THEN WE HAVE AN ERROR
00CC  0A C0                          OR      AL,AL           ;IS THE FIELD SIZE 0?
00CE  74 B3                          JE      OPTERR          ;YES - IT IS IN ERROR THEN
00D0  3C 0F                          CMP     AL,15           ;IS FIELD SIZE MORE THAN 15?
00D2  7F AF                          JG      OPTERR          ;YES - IT IS IN ERROR THEN
00D4  A2 0205 R                      MOV     TOKSIZ,AL       ;SAVE THE FIELD SIZE FOR JUSTIFIED FIELDS
00D7  EB 99                          JMP     SHORT OPT01     ;AND PROCESS FURTHER OPTIONS
00D9  E8 0247 R              OPTNUM: CALL    DECBIN          ;IS IT A NUMBER AFTER SLASH?
00DC  72 A5                          JC      OPTERR          ; NO - THEN IT IS AN ERROR
```

(Continued on next page)

Listing Two

```
00DE  0A C0                       OR      AL,AL           ;ZERO IS SPECIAL
00E0  75 07                       JNE     OPT08           ;NOT ZERO - SAVE IT IN ARRAY THEN
00E2  80 0E 0203 R 04             OR      FLAG1,F1ONE     ;ZERO MEANS ONE TOKEN PER LINE
00E7  EB 89                       JMP     SHORT OPT01     ;PROCESS SOME OTHER TOKEN THEN
00E9  2B C9                 OPT08:  SUB     CX,CX           ;GET A ZEROED DOUBLE REGISTER
00EB  8A 0E 0505 R                  MOV     CL,OUTCNT       ;GET OFFSET INTO OUTERS FOR THIS GUY
00EF  8D 1E 0506 R                  LEA     BX,OUTERS       ;POINT JUST BEFORE LIST OF OUTERS
00F3  03 D9                         ADD     BX,CX           ;BX POINTS TO ORIGIN 1 SAVE SPOT
00F5  88 07                         MOV     BYTE PTR [BX],AL ;SAVE THE TOKEN POSITION TO WRITE
00F7  FE 06 0505 R                  INC     OUTCNT          ;ADD ONE TO OUT COUNT
00FB  80 0E 0203 R 08               OR      FLAG1,F1SUB     ;MAKE SURE SUBSTITUTE FLAG IS ON
0100  E9 0072 R                     JMP     OPT01           ;AND PLAY IT AGAIN SAM
0103                        OPTIONS ENDP
                            ;
                            ;
                            ; NEXTOK - This subroutine will find the next token in the string pointed to
                            ;          by BX, with length contained in CX, and move it to the end of the
                            ;          token space.  An entry in TOKTBL will be created for this token.
                            ;          When the subroutine returns CX will be zero if the source data
                            ;          string is empty.  BX will point to the first character past the last
                            ;          token.
0103                        NEXTOK  PROC    NEAR
0103  8B 3E 0605 R                 MOV     DI,TOKPTR       ;GET POINTER TO WHERE TO PUT TOKEN
0107  8A 07                 NEXT01: MOV     AL,BYTE PTR 0[BX] ;LOOP PAST JUNK
0109  3C 20                         CMP     AL,' '          ;IS IT A LEADING SPACE?
010B  75 04                         JNE     NEXT03          ; NO - THEN WE HAVE A TOKEN
010D  43                            INC     BX              ;YES - POINT TO THE NEXT CHARACTER
010E  E2 F7                         LOOP    NEXT01          ;      AND TRY THAT ONE
0110  C3                            RET                     ;RETURN IF WE ARE DONE WITH THE OUTPUT
0111  8B F3                 NEXT03: MOV     SI,BX           ;SAVE POINTER TO START OF TOKEN
0113  B4 01                         MOV     AH,1            ;INITIAL GUESS FOR TOKEN LENGTH IS 1
0115  43                    NEXT04: INC     BX              ;POINT TO NEXT CHARACTER IN INPUT STRING
0116  8A 07                         MOV     AL,BYTE PTR [BX] ;GET THE CHARACTER
0118  3C 20                         CMP     AL,' '          ;IS IT THE END OF THE TOKEN?
011A  74 06                         JE      NEXT05          ;YES - WE HAVE SOME GOOD NUMBERS TO WORK WITH
011C  FE C4                         INC     AH              ; NO - INCREMENT COUNT OF CONTIGUOUS CHARACTERS
011E  E2 F5                         LOOP    NEXT04          ;CONTINUE TILL OUT OF CHARS OR A FIELD SEPARATOR
0120  FE CC                         DEC     AH              ;WE SHOULDN'T GET HERE BUT CORRECT FOR IT ANYWAY
0122  51                    NEXT05: PUSH    CX              ;SAVE NUMBER OF CHARACTERS LEFT IN SOURCE STRING
0123  F6 06 0203 R 03               TEST    FLAG1,F1RJ+F1LJ ;DO WE HAVE A MAXIMUM TOKEN LENGTH?
0128  74 2C                         JZ      NEXT09          ; NO - JUST A NORMAL TOKEN WRITE THEN
012A  3A 26 0205 R                  CMP     AH,TOKSIZ       ;YES - IS THIS TOKEN JUST RIGHT?
012E  74 26                         JE      NEXT09          ;IT SURE IS, WE WILL KEEP IT AS IS
0130  7C 06                         JL      NEXT06          ;IF TOKEN SIZE < MAX TOKEN SIZE WE MUST PAD
0132  8A 26 0205 R                  MOV     AH,TOKSIZ       ;OTHERWISE TAKE MAX TOKEN SIZE AS OUR OWN
0136  EB 1E                         JMP     SHORT NEXT09    ;AND CONTINUE NORMALLY
0138  A0 0205 R                NEXT06: MOV     AL,TOKSIZ       ;GET THE TOKEN SIZE WE MUST PAD TO
013B  2A C4                         SUB     AL,AH           ;AL NOW CONTAINS NUMBER OF SPACES NEEDED
013D  F6 06 0203 R 02               TEST    FLAG1,F1LJ      ;LEFT JUSTIFY THE THING? (PAD RIGHT WITH SPACE)
0142  74 0A                         JZ      NEXT07          ; NO - MUST PAD TO THE LEFT WITH SPACES
0144  A2 0204 R                     MOV     SPACES,AL       ;YES - SAVE HOW MANY SPACES TO FILL WITH
0147  80 0E 0203 R 10               OR      FLAG1,F1WORK    ;      MARK AS WORK TO DO LATER ON
014C  EB 08                         JMP     SHORT NEXT09    ;      AND JOIN MAINLINE CODE
014E  8A C8                 NEXT07: MOV     CL,AL           ;CX CONTAINS THE NUMBER OF LEADING SPACES
0150  C6 05 20              NEXT08: MOV     BYTE PTR [DI],' ' ;PUT A LEADING SPACE IN THIS TOKEN
0153  47                            INC     DI              ;POINT TO THE NEXT SLOT
0154  E2 FA                         LOOP    NEXT08          ;AND FILL IN ALL NEEDED SPACES
0156  8A CC                 NEXT09: MOV     CL,AH           ;CX NOW CONTAINS TOTAL NUMBER OF CHARS IN TOKEN
0158  FC                            CLD                     ;MAKE THE DIRECTION EVER UPWARD
0159  F3/ A4                        REP     MOVSB           ;MOVE THE TOKEN TO ITS SPOT
015B  F6 06 0203 R 10               TEST    FLAG1,F1WORK    ;IS IT LEFT JUSTIFIED (WE NEED SPACES)
0160  74 0F                         JZ      NEXT11          ; NO - WE ARE DONE WITH THE HARD PART THEN
0162  8A 0E 0204 R                  MOV     CL,SPACES       ;GET COUNT OF SPACES NEEDED
0166  C6 05 20              NEXT10: MOV     BYTE PTR [DI],' ' ;MOVE IN A TRAILING SPACE
0169  47                            INC     DI              ;POINT TO NEXT SLOT
016A  E2 FA                         LOOP    NEXT10          ;AND CONTINUE TILL ALL TRAILING SPACES DONE
016C  80 26 0203 R EF               AND     FLAG1,255-F1WORK ;RESET THE WORK TO DO BIT
0171  8B 16 0605 R          NEXT11: MOV     DX,TOKPTR       ;GET POINTER TO START OF THIS TOKEN
```

```
0175  89 3E 0605 R                      MOV     TOKPTR,DI       ;SAVE POINTER TO NEXT FREE TOKEN BYTE
0179  F6 06 0203 R 03                   TEST    FLAG1,F1LJ+F1RJ ;DO WE HAVE FIXED LENGTH TOKENS?
017E  74 04                            JZ      NEXT12          ; NO - TAKE THEM AS WE GET THEM
0180  8A 26 0205 R                      MOV     AH,TOKSIZ       ;YES - SET THIS TOKENS LENGTH TO WHATEVER
0184  8A CC                    NEXT12: MOV     CL,AH           ;SAVE LENGTH OF TOKEN
0186  B0 03                            MOV     AL,3            ;NUMBER OF BYTES PER ENTRY
0188  F6 26 0207 R                      MUL     TOKCNT          ;AX IS NOW AN OFFSET IN TOKTBL
018C  8D 36 0208 R                      LEA     SI,TOKTBL       ;POINT TO START OF TOKEN TABLE
0190  03 F0                             ADD     SI,AX           ;SI POINTS TO AN ENTRY IN TOKEN TABLE
0192  88 0C                            MOV     BYTE PTR [SI],CL ;MOVE IN LENGTH OF ENTRY
0194  89 54 01                         MOV     WORD PTR 1[SI],DX ;SAVE POINTER TO START OF TOKEN IN TABLE
0197  FE 06 0207 R                      INC     TOKCNT          ;COUNT ONE MORE TOKEN
019B  59                                POP     CX              ;CX CONTAINS NUMBER OF SOURCE CHARACTERS LEFT
019C  0B C9                             OR      CX,CX           ;DONE YET?
019E  74 01                             JZ      NEXT13          ;YES - RETURN
01A0  49                                DEC     CX              ; NO - CORRECT FOR UNDERCOUNTING BY ONE
01A1  C3                       NEXT13: RET                     ;AND RETURN
01A2                           NEXTOK  ENDP
                               ;
                               ;
                               ; WRITE - This routine will write the tokens to the standard output device.
                               ;         It is controlled by the settings of flags in FLAG1.
01A2                           WRITE   PROC    NEAR
01A2  2B C9                             SUB     CX,CX           ;GET AN EMPTY LOOP COUNTER
01A4  8A 0E 0207 R                      MOV     CL,TOKCNT       ;CL CONTAINS TOTAL NUMBER OF TOKENS READ
01A8  0B C9                             OR      CX,CX           ;DO WE HAVE ANYTHING TO WRITE OUT?
01AA  75 01                             JNZ     WRITE1          ;YES - THEN GO FOR IT
01AC  C3                                RET                     ; NO - WE ARE DONE BEFORE WE EVEN BEGIN
01AD  F6 06 0203 R 08          WRITE1: TEST    FLAG1,F1SUB     ;ARE WE CHANGING THEIR ORDER?
01B2  75 0B                             JNZ     WRITE3          ;YES - THEN USE DIFFERENT WRITE LOGIC
01B4  2A D2                             SUB     DL,DL           ; NO - JUST OUTPUT THEM ALL IN ORDER
01B6  E8 01E2 R                WRITE2: CALL    TOUT            ;WRITE THE SUCKER
01B9  FE C2                             INC     DL              ;POINT TO THE NEXT TOKEN
01BB  E2 F9                             LOOP    WRITE2          ;AND GO THROUGH THEM ALL
01BD  EB 18                             JMP     SHORT WRITE6    ;RETURN, A JOB WELL DONE
01BF  8A 0E 0505 R             WRITE3: MOV     CL,OUTCNT       ;GET THE NUMBER OF TOKENS TO WRITE
01C3  8D 1E 0506 R                      LEA     BX,OUTERS       ;POINT TO FIRST ONE TO OUTPUT
01C7  8A 17                    WRITE4: MOV     DL,BYTE PTR [BX] ;GET A TOKEN TO WRITE
01C9  3A 16 0207 R                      CMP     DL,TOKCNT       ;IS IT LESS THAN OR EQUAL TO MAX TOKEN?
01CD  7F 05                             JG      WRITE5          ; NO - DON'T WRITE IT THEN
01CF  FE CA                             DEC     DL              ;YES - ADJUST FOR ORIGIN ONE AND
01D1  E8 01E2 R                         CALL    TOUT            ;      WRITE THIS TOKEN THEN
01D4  43                       WRITE5: INC     BX              ;POINT TO THE NEXT TOKEN COUNT TO WRITE
01D5  E2 F0                             LOOP    WRITE4          ;AND LOOP THROUGH THE WHOLE LIST
01D7  F6 06 0203 R 04          WRITE6: TEST    FLAG1,F1ONE     ;ARE WE OUTPUTING ONE TOKEN PER LINE?
01DC  75 03                             JNZ     WRITE7          ;YES - THE LAST <cr><lf> HAS BEEN WRITTEN
01DE  E8 0213 R                         CALL    CRLF            ; NO - WRITE A TRAILING <cr><lf> SEQUENCE
01E1  C3                       WRITE7: RET                     ;DONE, GO HOME NOW
01E2                           WRITE   ENDP
                               ;
                               ;
                               ;
                               ; TOUT - This routine will find and write the token from the input line that is
                               ;        in position DL on that line.
                               ;
01E2                           TOUT    PROC    NEAR
01E2  53                                PUSH    BX              ;SAVE THE REGISTERS
01E3  51                                PUSH    CX
01E4  52                                PUSH    DX
01E5  B0 03                             MOV     AL,3            ;NUMBER OF BYTES PER TOKTBL ENTRY
01E7  F6 E2                             MUL     DL              ;GET AN OFFSET INTO TOKTBL FOR THIS ONE
01E9  8D 1E 0208 R                      LEA     BX,TOKTBL       ;POINT TO START OF TABLE
01ED  03 D8                             ADD     BX,AX           ;NOW POINT TO THE CORRECT 3 BYTE ENTRY
01EF  2B C9                             SUB     CX,CX           ;ZERO THE COUNTER
01F1  8A 0F                             MOV     CL,BYTE PTR [BX] ;GET THE NUMBER OF CHARACTERS IN THIS TOKEN
01F3  8B 57 01                          MOV     DX,WORD PTR 1[BX] ;AND POINT TO FIRST BYTE OF SAID TOKEN
01F6  E8 0226 R                         CALL    OSWRITE         ;WRITE TO THE STANDARD OUTPUT DEVICE
01F9  F6 06 0203 R 04                   TEST    FLAG1,F1ONE     ;ONLY ONE TOKEN PER LINE?
```

(Continued on next page)

Listing Two

```
0200  E8 0213 R                   CALL    CRLF            ;YES - WRITE A <cr><lf> SEQUENCE
0203  EB 0A                       JMP     SHORT TOUT03    ;      AND RETURN
0205  B9 0001                TOUT02: MOV     CX,1            ;LENGTH OF SPACE IS ONE
0208  8D 16 0A02 R                  LEA     DX,CHRSPA       ;POINT TO A SPACE
020C  E8 0226 R                   CALL    OSWRITE         ;WRITE TO STANDARD OUTPUT DEVICE
020F  5A                      TOUT03: POP     DX              ;RESTORE THE REGISTERS
0210  59                          POP     CX
0211  5B                          POP     BX
0212  C3                          RET                     ;AND RETURN
0213                          TOUT    ENDP
                              ;
                              ;
                              ;
                              ; CRLF - Everybody knows what this routine does.
                              ;
0213                          CRLF    PROC    NEAR
0213  50                          PUSH    AX              ;SAVE THE REGISTERS
0214  53                          PUSH    BX
0215  51                          PUSH    CX
0216  52                          PUSH    DX
0217  B9 0002                    MOV     CX,2            ;LENGTH OF CR LF STRING
021A  8D 16 0A03 R                LEA     DX,CHRCLF       ;POINT TO THE DATA TO WRITE
021E  E8 0226 R                   CALL    OSWRITE         ;WRITE TO STANDARD OUTPUT DEVICE
0221  5A                          POP     DX              ;RESTORE THE REGISTERS
0222  59                          POP     CX
0223  5B                          POP     BX
0224  58                          POP     AX
0225  C3                          RET                     ;AND RETURN
0226                          CRLF    ENDP
                              ;
                              ;
                              ; OSWRITE - This routine will write the characters pointed to by DS:DX, of
                              ;           length contained in CX, to the standard output file.  If any errors
                              ;           are detected a message will be written to the standard error device
                              ;           and flag F1EOJ will be set.
                              ;
0226                          OSWRITE PROC    NEAR
0226  BB 0001                    MOV     BX,1            ;HANDLE OF STANDARD OUTPUT DEVICE
0229  B4 40                      MOV     AH,40H          ;WRITE TO FILE OR DEVICE SYSTEM FUNCTION
022B  CD 21                      INT     21H             ;CALL THE OPERATING SYSTEM
022D  72 04                      JC      OSWR01          ;IF ERROR 5, OR 6 THEN END
022F  3B C8                      CMP     CX,AX           ;DID WE WRITE AS MANY CHARACTERS AS WE WANTED?
0231  74 13                      JE      OSWR99          ;RETURN IF ALL WENT WELL
0233  8D 16 09E1 R           OSWR01: LEA     DX,NOROOM       ;POINT TO THE NO SPACE MESSAGE
0237  B9 0021                    MOV     CX,LNOROOM      ;GET THE LENGTH OF THE MESSAGE
023A  BB 0002                    MOV     BX,2            ;GET HANDLE FOR STANDARD ERROR DEVICE
023D  B4 40                      MOV     AH,40H          ;WRITE TO FILE OR DEVICE SYSTEM FUNCTION
023F  CD 21                      INT     21H             ;LET HIM KNOW WE ERRED
0241  80 0E 0203 R 40            OR      FLAG1,F1EOF     ;PRETEND EOF ON INPUT DEVICE SO PROGRAM STOPS
0246  C3                    OSWR99: RET                     ;AND RETURN
0247                        OSWRITE ENDP
                            
                              ; DECBIN - On entry this routine has the first character to convert to binary
                              ;          in AL.  SI points to additional characters.  On exit SI points to
                              ;          the first non-numeric character found.  AL contains the binary value
                              ;          and Carry isn't set.  If Carry is set on return then an invalid
                              ;          number was found.
0247                        DECBIN  PROC    NEAR
0247  53                          PUSH    BX              ;SAVE A REGISTER
0248  E8 0273 R                   CALL    DECB04          ;CHECK FOR NUMERIC IN AL
024B  73 03                      JNC     DECB02          ;IF AL WAS NUMERIC IT IS NOW IN [0..9]
024D  F9                    DECB01: STC                     ;MAKE SURE CARRY FLAG IS SET
024E  5B                          POP     BX              ;RESTORE THE REGISTERS
024F  C3                          RET                     ;AND RETURN INDICATING AN ERROR
0250  8A D8                 DECB02: MOV     BL,AL           ;GET THE TOTAL SO FAR
0252  26: 8A 04                   MOV     AL,BYTE PTR ES:0[SI] ;GET A BYTE FROM THE INPUT STREAM
0255  46                          INC     SI
```

```
0256  E8 0273 R                     CALL    DECB04          ;CHECK IT FOR NUMERIC
0259  73 06                         JNC     DECB03          ;IF IT IS NUMERIC THEN JUGGLE SOME
025B  8A C3                         MOV     AL,BL           ;OTHERWISE GET THE VALUE TO RETURN
025D  F8                            CLC                     ;CLEAR THE CARRY BIT TO SAY IT WORKED
025E  4E                            DEC     SI              ;MAKE SURE NEXT CHAR TO GET IS NON-NUMERIC
025F  5B                            POP     BX              ;RESTORE THE REGISTER
0260  C3                            RET                     ;AND RETURN
0261  8A F8                 DECB03: MOV     BH,AL           ;SAVE THE NUMBER FOR A MINUTE
0263  B0 0A                         MOV     AL,10           ;GET THE BASE
0265  F6 E3                         MUL     BL              ;SHIFT CURRENT COUNT LEFT ONE POSITION (BASE AL)
0267  8A DF                         MOV     BL,BH           ;MAKE BX A GOOD NUMBER
0269  2A FF                         SUB     BH,BH           ;BX NOW CONTAINS 16 BIT VALUE OF ASCII DIGIT
026B  03 C3                         ADD     AX,BX           ;ADD IN THE LATEST DIGIT
026D  0A E4                         OR      AH,AH           ;MAKE SURE NO OVERFLOW
026F  75 DC                         JNE     DECB01          ;IF THERE WAS THIS IS AN ERROR
0271  EB DD                         JMP     SHORT DECB02    :CONTINUE ON OUR WAY
0273  2C 30                 DECB04: SUB     AL,'0'          ;IS IT LESS THAN A NUMBER
0275  7C 06                         JL      DECB05          ;YES - RETURN WITH CARRY SET
0277  3C 09                         CMP     AL,9            ;IS IT MORE THAN A NUMBER
0279  7F 02                         JG      DECB05          ;YES - RETURN WITH CARRY SET
027B  F8                            CLC                     ; NO - MAKE SURE CARRY IS OFF
027C  C3                            RET                     ;      THEN RETURN THE NUMBER
027D  F9                    DECB05: STC                     ;SET CARRY ON
027E  C3                            RET                     ;AND RETURN
027F                        DECBIN  ENDP
                            ;
                            ;
                            ; BUFGET - This routine will read one 'line' from the standard input device to
                            ;          BUFFER.  On exit CX contains the count of characters read.  F1EOF is
                            ;          set if an end of file condition is encountered.
027F                        BUFGET  PROC    NEAR
027F  50                            PUSH    AX              ;SAVE THE REGISTERS
0280  53                            PUSH    BX
0281  52                            PUSH    DX
0282  57                            PUSH    DI
0283  56                            PUSH    SI
0284  F6 06 0203 R 80              TEST    FLAG1,F1QEOF    ;SHOULD WE REFLECT AN IMMEDIATE EOF?
0289  74 0C                       JZ      BUFG00          ; NO - STANDARD LOGIC HERE THEN
028B  80 0E 0203 R 40             OR      FLAG1,F1EOF     ;YES - SET THE END OF FILE BIT
0290  80 26 0203 R 7F             AND     FLAG1,255-F1QEOF ;AND SAY IT IS NO LONGER PENDING
0295  EB 41                       JMP     SHORT BUFRET    ;RETURN NOW
0297  2B C9                 BUFG00: SUB     CX,CX           ;COUNT OF CHARACTERS GOTTEN
0299  8D 3E 0000 R                 LEA     DI,BUFFER       ;POINT DESTINATION TO BUFFER
029D  E8 02DE R              BUFG01: CALL    CGET            ;GET ONE CHARACTER
02A0  F6 06 0203 R 40             TEST    FLAG1,F1EOF     ;DID WE GET END OF FILE ON THAT TRY?
02A5  74 15                       JZ      BUFG02          ;YES - LETS HOPE IT IS AN ERROR
02A7  0A C9                       OR      CL,CL           ;IS THERE ANYTHING IN THE BUFFER?
02A9  74 2D                       JZ      BUFRET          ; NO - JUST RETURN WITH CX=0 AND F1EOF
02AB  B4 0D                       MOV     AH.0DH          ;YES - SLAP A <cr> ON THE END
02AD  E8 031C R                   CALL    CPUT            ;      PUT IT AT END OF BUFFER
02B0  80 26 0203 R BF             AND     FLAG1,255-F1EOF ;      CLEAR THE END OF FILE BIT
02B5  80 0E 0203 R 80             OR      FLAG1,F1QEOF    ;      SAY NEXT TIME TURN ON EOF FOR SURE
02BA  EB 1C                       JMP     SHORT BUFRET    ;      AND RETURN THIS LAST BUFFER
02BC  80 FC 0D               BUFG02: CMP     AH,0DH          ;IS IT THE RECORD TERMINATOR CHARACTER?
02BF  74 07                       JE      BUFG03          ;YES - DON'T TURN THAT INTO A SPACE
02C1  80 FC 20                    CMP     AH,20H          ; NO - IF NOT <cr> AND < 20H THEN MAKE A SPACE
02C4  7D 02                       JGE     BUFG03          ;IF >=20H THEN USER AS IT
02C6  B4 20                       MOV     AH,' '          ;OTHERWISE MAKE IT A SPACE
02C8  E8 031C R              BUFG03: CALL    CPUT            ;WRITE THE CHARACTER TO THE OUTPUT BUFFER
02CB  80 FC 0D                    CMP     AH,0DH          ;DID WE JUST WRITE THE RECORD TERMINATOR?
02CE  74 08                       JE      BUFRET          ;YES - THEN RETURN
02D0  80 F9 FF                    CMP     CL,255          ;HAVE WE WRITTEN 255 CHARACTERS YET?
02D3  75 C8                       JNE     BUFG01          ; NO - GET THE NEXT CHARACTER THEN
02D5  C6 05 0D                    MOV     BYTE PTR [DI],0DH ;MAKE IT A TERMINATOR
02D8  5E                     BUFRET: POP     SI              ;RESTORE THE REGISTERS
02D9  5F                            POP     DI
02DA  5A                            POP     DX
02DB  5B                            POP     BX
02DC  58                            POP     AX
02DD  C3                            RET                     ;AND RETURN
02DE                        BUFGET  ENDP
                            ;
02DE                        CGET    PROC    NEAR
```

(Continued on next page)

Listing Two

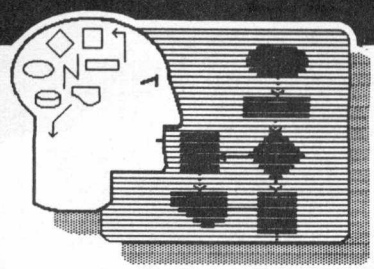
```
524     02DE  8B 16 0100 R              MOV     DX,GLEN         ;IS THERE ANY DATA IN GBUFF?
525     02E2  0B D2                     OR      DX,DX           ;IF COUNT IS ZERO THEN THERE ISN'T
526     02E4  75 25                     JNZ     CGET01          ;THERE IS DATA SO READ IT
527     02E6  51                        PUSH    CX              ;SAVE THE REGISTERS WE MIGHT NEED
528     02E7  57                        PUSH    DI
529     02E8  B4 3F                     MOV     AH,3FH          ;OS FUNCTION TO READ FROM STANDARD DEVICE
530     02EA  BB 0000                   MOV     BX,0            ;FILE HANDLE TO READ FROM (STANDARD DEVICE)
531     02ED  B9 00FF                   MOV     CX,255          ;NUMBER OF CHARACTERS TO READ
532     02F0  8D 16 0104 R              LEA     DX,GBUFF        ;POINT TO WHERE TO PUT THE DATA
533     02F4  89 16 0102 R              MOV     GBPTR,DX        ;SAVE POINTER TO FIRST CHARACTER
534     02F8  CD 21                     INT     21H             ;CALL THE OPERATING SYSTEM TO HANDLE THE WORK
535     02FA  A3 0100 R                 MOV     GLEN,AX         ;SAVE THE NUMBER OF CHARACTERS READ
536     02FD  8B D0                     MOV     DX,AX           ;PUT DATA COUNT IN DX WHERE IT BELONGS
537     02FF  5F                        POP     DI              ;RESTORE THE REGISTERS
538     0300  59                        POP     CX
539     0301  0B D2                     OR      DX,DX           ;DID WE GET DATA OR EOF?
540     0303  75 06                     JNZ     CGET01          ;DATA THIS TIME
541     0305  80 0E 0203 R 40           OR      FLAG1,F1EOF     ;SET THE END OF FILE ENCOUNTERED BIT
542     030A  C3                        RET                     ;AND RETURN
543     030B  8B 1E 0102 R      CGET01: MOV     BX,GBPTR        ;GET POINTER TO CHARACTER TO RETURN
544     030F  8A 27                     MOV     AH,BYTE PTR [BX]
545     0311  FF 06 0102 R              INC     GBPTR           ;GET CHARACTER AND INCREMENT POINTER
546     0315  FF 0E 0100 R              DEC     GLEN            ;DECREMENT LENGTH
547     0319  FE C1                     INC     CL              ;COUNT THIS CHARACTER
548     031B  C3                        RET                     ;AND RETURN THE CHARACTER
549     031C                    CGET    ENDP
550
551     031C                    CPUT    PROC    NEAR
552     031C  88 25                     MOV     BYTE PTR [DI],AH ;SAVE THE CHARACTER
553     031E  47                        INC     DI              ;POINT TO NEXT SPOT
554     031F  80 F9 FF                  CMP     CL,255          ;WILL WE OVERREACH NEXT TIME?
555     0322  75 01                     JNE     CPUT01          ; NO - GOOD THING
556     0324  4F                        DEC     DI              ;YES - CAN'T LET THAT HAPPEN
557     0325  C3                CPUT01: RET                     ;ALL DONE
558     0326                    CPUT    ENDP
559     0326                    CSECT   ENDS
560                                     END     MAIN

Segments and groups:

                N a m e                 Size    align   combine class

CSECT. . . . . .                0326    PARA    NONE    'CODE'
DSECT. . . . . .                0A05    PARA    NONE    'DATA'
STACK. . . . . .                00C0    PARA    STACK   'STACK'

Symbols:

                N a m e                 Type    Value   Attr

BUFFER . . . . . .              L BYTE  0000    DSECT   Length =00FF
BUFG00 . . . . . .              L NEAR  0297    CSECT
BUFG01 . . . . . .              L NEAR  029D    CSECT
BUFG02 . . . . . .              L NEAR  02BC    CSECT
BUFG03 . . . . . .              L NEAR  02C8    CSECT
BUFGET . . . . . .              N PROC  027F    CSECT   Length =005F
BUFRET . . . . . .              L NEAR  02D8    CSECT
CGET . . . . . .                N PROC  02DE    CSECT   Length =003E
CGET01 . . . . . .              L NEAR  030B    CSECT
CHRCLF . . . . . .              L BYTE  0A03    DSECT
CHRSPA . . . . . .              L BYTE  0A02    DSECT
CPUT . . . . . .                N PROC  031C    CSECT   Length =000A
CPUT01 . . . . . .              L NEAR  0325    CSECT
CRLF . . . . . .                N PROC  0213    CSECT   Length =0013
DECB01 . . . . . .              L NEAR  024D    CSECT
DECB02 . . . . . .              L NEAR  0250    CSECT
DECB03 . . . . . .              L NEAR  0261    CSECT
DECB04 . . . . . .              L NEAR  0273    CSECT
DECB05 . . . . . .              L NEAR  027D    CSECT
```

```
DECBIN . . . . . N PROC  0247   CSECT   Length =0038
F1EOF. . . . . Number  0040
F1LJ . . . . . Number  0002
F1OERR . . . . . Number  0020
F1ONE. . . . . Number  0004
F1QEOF . . . . . Number  0080
F1RJ . . . . . Number  0001
F1SUB. . . . . Number  0008
F1WORK . . . . . Number  0010
FLAG1. . . . . L BYTE  0203   DSECT
GBPTR. . . . . L WORD  0102   DSECT
GBUFF. . . . . L BYTE  0104   DSECT   Length =00FF
GLEN . . . . . L WORD  0100   DSECT
LNOROOM. . . . . Number  0021
MAIN . . . . . F PROC  0000   CSECT   Length =006A
MAIN00 . . . . . L NEAR  001A   CSECT
MAIN01 . . . . . L NEAR  0038   CSECT
MAIN02 . . . . . L NEAR  0055   CSECT
MAIN03 . . . . . L NEAR  0061   CSECT
MSGVER . . . . . L BYTE  098B   DSECT
NEXT01 . . . . . L NEAR  0107   CSECT
NEXT03 . . . . . L NEAR  0111   CSECT
NEXT04 . . . . . L NEAR  0115   CSECT
NEXT05 . . . . . L NEAR  0122   CSECT
NEXT06 . . . . . L NEAR  0138   CSECT
NEXT07 . . . . . L NEAR  014E   CSECT
NEXT08 . . . . . L NEAR  0150   CSECT
NEXT09 . . . . . L NEAR  0156   CSECT
NEXT10 . . . . . L NEAR  0166   CSECT
NEXT11 . . . . . L NEAR  0171   CSECT
NEXT12 . . . . . L NEAR  0184   CSECT
NEXT13 . . . . . L NEAR  01A1   CSECT
NEXTOK . . . . . N PROC  0103   CSECT   Length =009F
NOROOM . . . . . L BYTE  09E1   DSECT
OPT01. . . . . L NEAR  0072   CSECT
OPT02. . . . . L NEAR  007B   CSECT
OPT04. . . . . L NEAR  0089   CSECT
OPT05. . . . . L NEAR  00A7   CSECT
OPT06. . . . . L NEAR  00B7   CSECT
OPT08. . . . . L NEAR  00E9   CSECT
OPT4A. . . . . L NEAR  0093   CSECT
OPT6A. . . . . L NEAR  00C3   CSECT
OPTERR . . . . . L NEAR  0083   CSECT
OPTIONS. . . . . N PROC  006A   CSECT   Length =0099
OPTMGL . . . . . Number  0021
OPTMSG . . . . . L BYTE  09C0   DSECT
OPTNUM . . . . . L NEAR  00D9   CSECT
OSWR01 . . . . . L NEAR  0233   CSECT
OSWR99 . . . . . L NEAR  0246   CSECT
OSWRITE. . . . . N PROC  0226   CSECT   Length =0021
OUTCNT . . . . . L BYTE  0505   DSECT
OUTERS . . . . . L BYTE  0506   DSECT   Length =00FF
SPACES . . . . . L BYTE  0204   DSECT
THREE. . . . . L BYTE  0206   DSECT
TOKCNT . . . . . L BYTE  0207   DSECT
TOKENS . . . . . L BYTE  0607   DSECT   Length =0384
TOKPTR . . . . . L WORD  0605   DSECT
TOKSIZ . . . . . L BYTE  0205   DSECT
TOKTBL . . . . . L BYTE  0208   DSECT   Length =02FD
TOUT . . . . . N PROC  01E2   CSECT   Length =0031
TOUT02 . . . . . L NEAR  0205   CSECT
TOUT03 . . . . . L NEAR  020F   CSECT
WRITE. . . . . N PROC  01A2   CSECT   Length =0040
WRITE1 . . . . . L NEAR  01AD   CSECT
WRITE2 . . . . . L NEAR  01B6   CSECT
WRITE3 . . . . . L NEAR  01BF   CSECT
WRITE4 . . . . . L NEAR  01C7   CSECT
WRITE5 . . . . . L NEAR  01D4   CSECT
WRITE6 . . . . . L NEAR  01D7   CSECT
WRITE7 . . . . . L NEAR  01E1   CSECT

Warning Severe
Errors  Errors
0       0
```

End Listing

THE SOFTWARE DESIGNER

A Realizable Fantasy



by Michael Swaine

In 1976, Jim Warren, the guiding spirit of this magazine's early days, defined in Jim Warren style the term *realizable fantasies*: "... projects that we feel (1) are within the bounds of current technology and knowledge, (2) can be implemented by members of the hobbyist community, and (3) can ... be realized within 24 months or less."

Warren asserted *DDJ*'s commitment to pursue and promote such realizable fantasies, and he pointed to their probable source: "You are part of this ... the proposals and designs and certainly the implementations will come from you. Send us your ideas, your creations, your problems, and your solutions, so that we may share them with everyone."

DDJ was a hobbyist newsletter in 1976, and today it's a magazine read by professional programmers, an evolution that closely parallels the evolution of the microcomputer software industry. One regrettable consequence of all this seriousness is a dearth of fantasy: little fantasy in *DDJ*, little fantasy in the industry, little fantasy in the design of new software. If you are keeping an eye on the bottom line, you aren't looking up.

Feeling as I did that Warren's spirit of playful creativity was ominously missing in software design in the mid-80s, I hit upon the simple plan of calling some active software designers and others who had thoughts about software design and asking them what tool of the trade they felt was missing: the realization of what fantasy would make the task of software design easier, more enjoyable, more accessible to a new generation of programmers, more manageable to those currently doing design work?

As I say, it was a simple plan—simple to the point of naivete as it turned out. I soon found, as I began polling people for their realizable fantasies, that the question was not as open-ended as I thought. It appears to have one answer.

Here's Steve Dompier, software designer: "I want to program by flowcharting. I want to draw a flowchart and employ a tool that turns the flowchart into a program. It might produce source code or object code—it doesn't matter. We already have menus of icons; why not icons for flowchart symbols? Function boxes, decision boxes ... we could have icons for the various loop structures.

"Maybe the tool would work hierarchically: you'd start with a simple design then get more and more detailed. That implies the top-down design approach, which is the way at least some programmers work. And you could examine the structure you were building hierarchically: point to a box in the structure, zoom in on it, and it expands to show what's in it.

"Building at the lowest level, you might have to type some code, but you'd certainly have some library functions available, represented as icons, selectable by pointing. Looking at the program, you should be able to zoom down to a specific area, to a specified level, even right down to source code."

Here's Lee Felsenstein, hardware designer: "What I'd like to see—quite literally see—is the ability for the visual representation of structure, very much like a schematic diagram, but for software. I'd like to see software schematic diagrams realized in computer graphics. It's been done for the hardware with CAD systems."

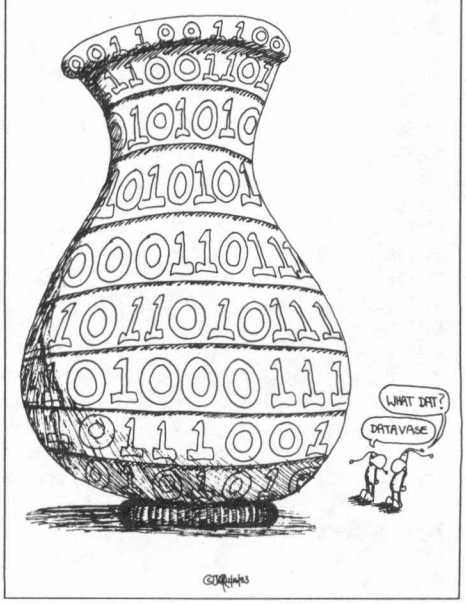
It soon became clear that the ability to design by manipulating visual representations of program structures is something that very many designers want. That's the fantasy. Is it realizable? Sure. Should it be realized? Are you kidding? That this has not yet happened can perhaps be relegated to the conventional-wisdom realm of explanation, subdomain barefoot progeny of cobblers.

The software design tool described here could be in the process of development now; someone reading this column could at this moment be working on it. Or could start work on it tomorrow or today. This minute.

The challenge is implicit. But let's be blatant. Christmas is coming. The software industry, after all, has been good to a lot of people. It has given the world (or that part of it with access to computers at least) some very useful tools. Won't someone give the software industry a present?

(There's a second challenge implicit in all this. Despite my pretense that a single answer exists to the question of what software design tools are needed, fantasy is inherently unbounded. There are, of course, infinitely many realizable fantasies of software design. You are welcome, as ever, to send us yours.)

DDJ



COMPUTER CALISTHENICS

by Michael Wiesenberg

Mid-morning at I–Q Industries in the heart of Silicon Valley. Grey Scrivener, senior technical writer specializing in software manuals, starts a fresh pot of coffee brewing at one of I – Q's permanent coffee stations. He glances up at the convex mirror mounted near the ceiling where the corridor intersects another. Seeing someone heading his way, Scrivener hurriedly adjusts the knot in his bottle-green knit tie.

Spotswood Gilbert, senior programmer, rounds the corner carrying a chocolate-covered doughnut. "Mornin', Grey. Sure glad you're makin' some *good* coffee. Can't stand that company stuff."

"Good morning, Spotswood. The coffee will be ready in a few minutes. How are you today?"

"Fine. How's the novel comin'?"

"I'm on the third rewrite."

Bob Levin, hotshot programmer, ratty sneakers squeaking, approaches the coffee machine. Scrivener casts a faintly disapproving glance at the young man's uncombed blond hair, trailing over a wrinkled Star Wars t-shirt. Levin stops. "How ya doing, Spotty? Hey, Grey."

Gilbert smiles. "Howdy, Bobby. You're up kinda early, aren't you?"

"I left early last night, so I thought I'd make it in for the morning doughnuts."

The water has dripped through. Scrivener fills Gilbert's cup and then his own. "By 'early' he means he left before midnight."

Sally McRae, another hotshot programmer, arrives wearing her usual "uniform": jeans, Nikes, and a Bay-to-Breakers t-shirt. "Hi, guys. You're here early, Bob. Don't tell me *you* made the coffee."

Levin looks embarrassed. "Nah, I don't know how to run that thing. Grey made it."

Scrivener sniffs. "Something's on fire."

McRae jumps. "What? Where?"

Marian Smith, system operator, appears, cigarette hanging out of her mouth. Scrivener fans the air with his hand to dissipate the smoke. Smith takes no more notice of this than she does the other criticism she frequently gets about her smoking. "Good morning, gentlemen, Sally. Who's on Goldfinger? I have to bring it down this afternoon. I'll be backing up all the system disks, but you'll have to take care of your own privates."

Levin eyes McRae appreciatively. "Playing two-man volleyball at noon, Sally?"

"Yes, and I don't see why it has to be called two-*man*. What's wrong with two-*person* volleyball?"

Scrivener always has the full story on word usage. "In a phrase like 'two-man volleyball' or a word like 'chairman,' the *man* has no gender. It refers to both men *and* women. When historians refer to the history of *man*, they are not excluding women, you know."

McRae looks him right in the eye. "Crap."

Gilbert looks uncomfortable. Where he's from, ladies don't talk that way. "Say, I got an interesting message on IQMAIL today. It was from my friend at HP, Jim Davis. You know, he's the one into puzzles. He gave me a great one. He said it was an old one he'd found that could be worked out by hand, but he thought a computer could do it a lot more efficiently. I wonder if you could write a program to solve it, Bob?"

"Let's hear it. I'll look at it after work."

McRae turned her scorn from Scrivener to Gilbert. "What makes you think *I* can't do it?"

"Well, sure, let's all give it a try. Here it is:

"There are two world-famed mathematicians, Mr. P and Mr. S. A friend of theirs who likes puzzles tells both of them that he has two numbers in mind. The only thing he will tell both of them about these two numbers is that they are both integers, greater than 1, and they are not the same.

"He then whispers to Mr. P the product of the two numbers.

"He whispers to Mr. S the sum of the two numbers.

"Mr. P knows that Mr. S knows the sum, but he doesn't know what that sum is. Similarly, Mr. S knows that Mr. P knows the product, but he doesn't know what that product is.

"The following conversation then takes place:

"Mr. P: 'I don't know the numbers.'

"Mr. S: 'I know that; I also don't know the numbers.'

"Mr. P: 'Oh, then I know the numbers.'

"Mr. S: 'Really, then I do also.'

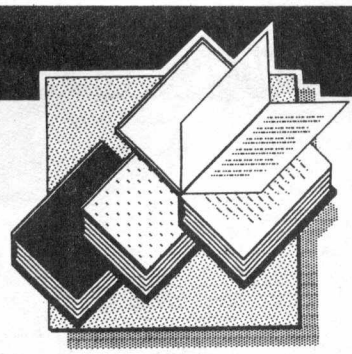
"Your task, of course, is to discover the two numbers. It would take you a long time to figure them out by hand, so write a program to do it. A *short* program."

Smith is more interested in IQMAIL than puzzles. She sets her cigarette down next to the coffee maker, its lit edge extending beyond the wood veneer top of the table, and pours herself another cup. "Is IQMAIL working for everybody? I've heard complaints from some of the other divisions."

Levin also pours a second cup. "I patched in my own communications package for a backup when IQMAIL goes down, which was twice yesterday. That's a terrific puzzle, Spots. The smallest two numbers that fit are 2 and 3, right?"

"Yes. You can see that their product is 6 and their sum is 5. But these two gentlemen are expert mathematicians. They would know that the numbers

(Continued on page 124)



Programming in C

by Stephen G. Kochan
Published by Hayden Book Company
373 pages

Reviewed by Ian Ashdown

There are now as many books about the C programming language as you could possibly want—from simple tutorials about the basics of the language to a completely documented implementation of a C compiler . . . and of course, that definitive work, Kernighan & Ritchie's *The C Programming Language*.

C is a programming language noted for its sparseness and economy of expression. Most books on C are relatively slim, reflecting its powerful simplicity. Not so, however, *Programming in C*; at 373 pages, this book is nothing short of voluminous. Each subject is dealt with in exhaustive and ultimately exhausting detail. If you are in search of an answer about almost any aspect of C, you will find it in this book. You will also find it quickly—the index is thorough and well organized.

The promotional blurb on the cover reads: "A complete, easy-to-understand introduction to the C programming language for the novice and experienced programmer alike. Contains over 90 program examples and covers all the latest language features." Unlike most promotional claims, this is one that I fully agree with. The book starts with a very quick review of programming fundamentals, shows you how to write, compile, and execute a simple C program, and then is off and away into examining the language itself.

What do you want to know about? Variables, constants, data types, and arithmetic expressions? Program flow control? Arrays, functions, structures, character strings, pointers, bit operations, and bit fields? The preprocessor, perhaps, or separate compilation? Input and output or maybe miscellaneous and advanced topics such as unions, dynamic memory allocation, and the goto statement (more on this later)? *Programming in C* covers them all with many clear and detailed examples.

The exhaustive detail of this book can best be demonstrated by an example. In *any* language, how can one possibly find enough to say about the lowly if statement to fill 18 pages of text? Answer: Read this book and see.

I must remind myself, however, that I am reading the book as a fairly proficient C programmer. It wasn't too long ago that I taught myself the language with only K & R's book as a guide. It was a frustrating experience at times, compounded by that book's sparse literary style. For the neophyte, *Programming in C* is undoubtedly a far better book from which to learn the language.

Having learned C, I have found no better reference book for my tastes than K & R's *The C Programming Language*. There are days, however, when I simply cannot remember some of C's more cryptic features, and K & R does nothing to disperse the mental fog. Over the past month, I have found that *Programming in C*, despite the author's propensity for verbosity, serves very nicely. In other words, the book truly is "for the novice and experienced programmer alike."

A full C compiler running under Unix on a minicomputer is assumed throughout the book, with only passing references to the idiosyncracies of microcomputer versions. This approach has obvious advantages in that the book itself will stand as a valid reference manual for as long as the C language is in general use. The same cannot be said for some other books on C that are based on current versions of particular C compilers.

Other nice features of this book are its appendices. Appendix A is a summary of the language, written in a style that is both concise and clear. (This is in marked contrast to K & R's Appendix A, "The C Reference Manual." Their language summary is concise but rarely clear to the average reader.) Especially useful is a full range of examples involving pointers, with explanations in English.

Appendix B, entitled "Common Programming Mistakes," is unfortunately too short to be of much use. But, then, serious programming in C means that you likely will make every mistake possible—the power and conciseness of the language guarantees it. Learn by doing and have a good reference manual by your side at all times to rescue you.

Appendix E discusses a program that all microcomputer-bound C programmers dream of: "lint." The discussion, which is a short page and a half, merely covers what this program analyzer looks for when presented with C source code. It flags such potential problems as undeclared variables, unreachable code, improperly declared functions, invalid structure member accesses, possible operator precedence errors . . . sigh. What I wouldn't give to have the source code for this program!

K & R's book is often referred to as the standard reference manual for the language, and yet it does not cover some extensions that have since become standard for Unix-based C compilers. You can now pass structures as arguments (not just pointers to them) to functions, and a new data type called "void" explicitly tells the compiler that a particular function does not return any value. Also the "enumerated" data type, taking a cue from Pascal and other languages, allows a restricted range of values to be assigned to a variable. *Programming in C* is one of the few books that discusses

these features in detail. Now if only the microcomputer C compiler writers could take heed of these extensions . . . .

The author does display some interesting programming prejudices. The goto, break, and continue statements are relegated to the back of the book in a chapter entitled "Miscellaneous Features and Advanced Topics." The reasoning is that these statements serve only to "obfuscate the program's logic and are not generally considered a part of good programming practice."

While I agree with the author that the goto statement is an anachronism that should be avoided whenever possible, I cannot agree with his argument about the break and continue statements. I find that these two statements serve to simplify and clarify the logic of program loops.

Also curious is the relegation of command line arguments to the same chapter. This to me is one of the most powerful features of C—it brings some of the grandeur of Unix to CP/M-based microcomputers, among other things. It also greatly extends the utility of programs that present the user with runtime options. It should have been given more space than its allotted page and a half.

Apart from that, I can say, while the book may be too wordy for my liking, it nevertheless can serve as an excellent reference on the C language as well as a tutorial for learning C. *Programming in C* may not be the ideal book for your needs, but by all means give it your consideration if you see it.

Pascal for Programmers

by Olivier Lecarme and Jean-Louis Nebut
Published by McGraw-Hill
$22.95, 272 pages
Reviewed by Dr. Joseph B. Rothstein

The position that BASIC has long enjoyed as the language of choice among beginning programmers faces a serious challenge from Pascal. Designed by Kathleen Jensen and Niklaus Wirth in the early 1970s, the Pascal programming language was intended to implement the principles of structured programming that were emerging at that time.

Because the design of the language encourages good programming habits, Pascal has been adopted for use in the first course in programming at many colleges and universities, and it is now available on most microcomputers as well.

Numerous texts use Pascal to introduce the principles of computers and programming to the neophyte, and some of them serve this audience well. However, there is a need for a text that would introduce and describe the entire Pascal language to someone already familiar with computer programming; it is this need that Lecarme and Nebut address in *Pascal for Programmers*. Although Pascal is well suited to beginners, it is also a powerful and elegant general-purpose language that has achieved considerable acceptance among business and scientific programmers as well.

Pascal for Programmers attempts to discuss each aspect of Pascal for experienced programmers—that is, those who don't need explanations of what computers, programs, algorithms, or expressions are. The authors have assumed that readers are already familiar with some high-level programming language, such as Fortran, BASIC, or PL/I, and that they understand the fundamental concepts of computer programming.

As a result, beginning programmers may find the text inappropriate and confusing, and the authors make it clear that the book isn't intended for use in an introductory course in computer programming. Rather it offers an exhaustive treatment of Pascal and a precise description of the language standardized by ISO (International Standards Organization). As such, it differs from the many nonstandard implementations that attempt to address Pascal's widely acknowledged limitations in such areas as string manipulation and file handling.

Given its narrow focus on the needs of experienced programmers and its strict adherence to the ISO standard, *Pascal for Programmers* serves its intended audience superbly. The first chapter offers a brief introduction and even briefer history of Pascal, followed by practical suggestions on how to use the book. Chapters 2 through 5 present the basic building blocks that constitute Pascal programs. The remaining ten chapters present the different structures that enable the programmer to construct complete, complex programs from these basic building blocks.

Each chapter deals with one particular programming construct, relating to either data structures or the implementation of algorithms. Although each chapter is relatively independent of the others, they are presented in an orderly sequence that introduces the most difficult concepts toward the end. For the programmer already familiar with Pascal, this facilitates quick reference to a complete discussion of a particular construct, without the need to leaf through the entire book.

Examples are used liberally throughout. These range from simple, short procedures to complete, complex, and lengthy programs, depending on the topic under discussion. In this regard, the text is far superior to others that use only a syntax chart and a statement or two to illustrate a point. While such an approach may focus the reader's attention on the construct under consideration, it does not offer the broader view used throughout *Pascal for Programmers*. The examples are generally followed by remarks that amplify and add to the sense of the code itself. As the book progresses, both the examples and the accompanying remarks become longer and more complex, as Pascal's power and elegance are more fully demonstrated.

Exercises are included as well, but they are not the true/false or short-answer variety commonly found in more introductory tests. Rather they require complete and challenging programs or program fragments. An appendix offers solutions to one exercise selected from each chapter. Other appendices include collected syntax diagrams, a bibliography, tables and reference lists, and a discussion of implementation-defined and implementation-dependent features.

More and more programmers and educators are coming to believe that, once one understands the fundamental principles of programming, any given language represents only a particular embodiment of that relatively fixed set of basic constructs. Conversely, if the student lacks an integrated view of programming, each new language will pose a seemingly new challenge, unrelated to

other languages learned previously.

Pascal for Programmers seems written with the broad view in mind. While it does not directly contrast Pascal code with that of other languages, high-level concepts are discussed from the perspective of language independence before the particulars of their embodiment in Pascal are introduced.

While the text is inappropriate to the needs, background, or budget of the curious beginner, its thorough, practical focus on the essence of Pascal qualifies it as a worthwhile continuing-education and reference tool for the programmer. If you want to teach yourself Pascal in a few evening's time or brush up on its more complex features without wading through background or introductory material of secondary importance, *Pascal for Programmers* is perhaps the best such text available today.

Microcomputer Software Design

by Sally Campbell
Published by Prentice-Hall
$12.95
ISBN: 0-13-580621-6
Reviewed by Richard L. Lozes

First, a disclaimer: Microcomputer software design is no different than any other software design. However, because microcomputers are also consumer items, the title appeals to a mass market. How well does the book address its topic in that market?

As the author admits in her preface, professional programmers write and study texts of greater breadth and depth. Indeed, an example she chose to develop throughout the book is amusing. I doubt that anyone with only the study of this book could cost effectively produce a correct payroll system. Such software is much better purchased.

Ms. Campbell surveys many of the techniques used in commercial software creation, all of which were invented in the sixties and seventies. Hierarchical Input-Process-Output (HIPO) charts, pseudocode, flowcharts, modularization, and testing are all discussed.

The first chapter could well have been left out. It is no more than a museum catalog of computer parts, imparting no understanding of software design. (For instance, whether a computer has a motherboard really has no bearing on how it is programmed.) The second chapter, on problem definition, glosses over the specification phase in its haste to begin design work. Such haste is well known to be fatal to programming efforts.

The author ignores two of the overriding characteristics of software creation as it is currently practiced: programming is much more of a social activity, and programming is very expensive. The expense is the reason for the existence of the techniques discussed. And, true, individual programmers *do* use techniques, but they also participate in reviews, walk-throughs, and heated discussions. One would like to see some acknowledgement of this "engineer's" viewpoint as motivation.

Microcomputer Software Design can be recommended only to those with some knowledge of the programming process who desire a shallow introduction to HIPO, pseudocode, file definitions, or debugging. Otherwise, $12.95 can be better spent elsewhere.

Self-Organization and Associative Memory

by Teuvo Kohonen
Published by Springer-Verlag
ISBN: 0-387-12165-X
Reviewed by Richard L. Lozes

Associative memory is a behavior peculiar to certain adaptive filters. If x denotes a key, M a set of (memory) parameters, and y data to be recalled, then

$$y = y(x,M)$$

represents associative recall if M = M (historical y).

Professor Kohonen develops his book around this type of function, seeking the conditions under which memory is able to store compressed data and to reexpand it for recall. He pursues notions of input-driven system reorganization, devoting a chapter to "self-organizing feature maps."

Many of us in the computer field are familiar with content-addressable memory. Such a memory is characterized by key-data pairs that are stored as received, one pair per location. Recall is effected by specification of a key alone and involves only the word(s) holding the requested key. This simple, direct, one-to-one mapping is a good deal less general than the associations that Professor Kohonen discusses.

Associative memories are characterized by a distribution of data throughout the memory medium, similar to the way in which a hologram stores data, but they also possess key-based retrieval the way content-addressable memories do. Mathematically, this distributed memory is reflected in the changing of the entire set of parameters, M—thus, the notion of a system reorganization. This is the type of memory that typifies the book. Clearly, the author is really attempting to answer the enigma of biological memory in preference to the well-understood technological memories.

The second chapter surveys and summarizes the necessary mathematical tools. These are drawn almost entirely from the domain of linear mathematics—in particular, from linear algebra. Nonlinearity, not being required at the lowest levels (within each memory element, say), is deemphasized. Actually, feedback via the associative parameters, M, above suffices to introduce the memory behavior. This chapter is particularly easy to read, despite the breadth of its coverage. In good European tradition, the author has amply referenced the original literature.

I must level one serious criticism against this monograph. Professor Kohonen refuses to permit arbitrarily complex memory elements. Rather he insists that they be "physical." I believe that he is mistakenly assuming that "physical" systems are inherently and economically describable by simple mathematics. That strikes me as naive. On the one hand, one can always reduce complex systems to simpler components, either exactly or approximately. On the other hand, nothing indicates that biological memory is characterized by (mathematically) simple cells. In short, while Occam's razor demands simplicity, Professor Kohonen apparently has shaven away far more than fuzziness.

Nevertheless, as an introduction to the field of associative memory, this little book is quite reasonable. A reader of modest mathematical prowess will receive sufficient inspiration and

information here to start into the field.

Interactive Programming Environments

Edited by David R. Barstow, Howard E. Shrobe, and Erik Sandewall
Published by McGraw-Hill
$34.95, 609 pages

Reviewed by Tom Provenzano

A great deal of research is in progress in the area of automated software engineering environments—what some might call expert systems for programmers. Software engineering environments provide an integrated set of tools to assist the programmer during most phases of the software development process, from system design to software testing. *Interactive Programming Environments*, edited by Barstow, Shrobe, and Sandewall, is a collection of 28 papers describing the research under way and the philosophy behind the building of programming environments.

The book is divided into five sections that explore issues and problems in the construction of interactive programming environments (IPSEs). The editors admit in the preface that "programming environment" does not have a generally accepted meaning, but they offer a definition of "computer-aided design systems for software."

The papers in the book focus on interactive as opposed to batch types of environments, because the writers feel that a dialogue form of interaction between man and machine is the optimal method of implementing and using software engineering environments. None of the papers addresses automated software production—programs that write programs—since that is an entirely separate field of its own, heavily submerged in artificial intelligence research. What is addressed in these papers is the middle ground between manual systems (the way software is currently developed) and automated systems, namely IPEs.

The first section in the book, "Perspectives on Interactive Programming Environments," consists of three papers exploring the motivations and reasons for developing IPEs. The first paper, "Breaking the Complexity Barrier (Again)," by Terry Winograd of Stanford University, written in 1973, sketches what an IPE should be like and what it must do to help programmers handle the complexity involved in building modern software systems. Winograd describes an integrated system of tools containing knowledge about the system being built and acting as a sort of "programmer's assistant" by providing precompilation error checking, system question answering, and debugging support throughout all stages of the software development process. Most of the other papers in the book describe systems that attempt to implement what Winograd outlines in this early paper.

One major issue that becomes apparent after reading the papers in this book is that of the difference between the incremental growth vs. the life cycle approach to software development. Incremental growth is akin to the software prototyping approach of building a working base and adding enhancements piece by piece, whereas the life cycle method follows a more rigid and inflexible structure of design/code/test. Because most of the papers are authored by people involved in academic or research efforts, the former approach seems to be preferred. A question to keep in mind when reading these papers is: Is the incremental growth method of software development necessarily superior to the life cycle method, especially for systems built in a nonacademic, industrial environment?

Other IPE features discussed throughout the book include data-driven systems, smart editors and debuggers, and DWIM ("Do What I Mean, Not What I Say") facilities. A primitive example of the last-named feature is having the system overlook simple typing errors and perform the command closest to what the user intended (after querying to ensure it's OK to do so).

Included in this volume are papers describing programming environments based on LISP (INTERLISP, LISP Machine, the Programmer's Assistant), C (Unix, Programmer's Workbench), Pascal (Pathcal), Ada (APSE—Ada Programming Support Environment), and smart editors (EMACS, DED, MENTOR). The book's final section, "The Future of Interactive Programming Environments," contains another paper by Terry Winograd in which he conjectures that the next step in IPEs must be the elimination of the necessity of having programmers work with specific programming languages. Instead they will be developing software at a level conceptually above high-order languages. Systems will be built out of software components, and the programming environments used to build such systems will know a great deal about software and software development in general.

Interactive Programming Environments is a good source of information on the continuing research in a field that shows promise in improving and altering the way software systems are developed. The book is fairly easy reading (a knowledge of LISP is helpful but not essential for some of the papers) for those with software engineering backgrounds and will be much appreciated by those implementing or interested in automating the software production process.

I would hope, though, that all implementors of future IPEs might take a hint from the Emily system described in the book and provide a "sympathy" button, as explained by Wilfred J. Hansen: "When the user is frustrated he can push a sympathy button. In response, Emily displays at random one of ten sympathetic messages." We can all use a pat on the shoulder from our digital friends now and then.

Talking Chips: IC Speech Synthesis

by Nelson Morgan
Published by McGraw-Hill
178 pages, 150 illustrations,
$24.50

Reviewed by Dennis Cashton

One of the fastest growing areas of technology for computers is speech synthesis. But it is not a new area. Work on this electronic marvel has been going on since 1939 when a device called VODER was introduced. If this subject holds the least bit of interest for you, this work is an excellent addition to your library.

At first glance, you might think that unless you are an engineer this book is too technical. Upon further examination, however, you would find that it is entertaining and not at all intimidating. To be sure, this is not a "how to" book but more of an explanation of

how speech synthesis started, how it works, who's doing it, and maybe where it will go in the future.

Nelson Morgan has written a serious book, but this has not stopped him from making it very amusing. He makes occasional references to some very silly uses for speech synthesis, the first of which is his mythical "talking toaster." There are numerous cartoons depicting equally silly situations with speaking devices. Even with all the frivolity, he never fails to stay on target with his writing. He starts out with a description of what speech synthesis really is then gives a basic overview of the several ways it can be accomplished. Continuing on through the hardware, he gets to a description of how some of today's speech synthesis chips work.

The next part of the book is where things get a little deep for the nontechnical person. This is where Mr. Morgan gets into the actual analysis of speech, including waveforms, estimation, classification, and pitch tracking (to drop a few names). He then goes on to discuss synthesis by rule and spends an entire chapter on the subject of obtaining quality audio to use for creating synthesized speech.

The final chapter covers how to send your own speech to an IC manufacturer and have it put on a chip, new ideas for speech synthesis, music and sound effects, and a section on speech recognition. After reading this last section you will understand why a lot more work has to be done in this area.

On the whole, the author does a wonderful job of presenting his topic. He compares the different techniques for speech analysis and synthesis, and he points out why some companies are using some methods and others are using different ones. If waveforms are not your thing, but you are interested in the concept of speech synthesis, it is quite simple to skip right over the more technical sections and still get a lot out of the book. But if you like formulas and models and such, the book is filled with them. There's even an appendix that describes Fourier analysis and windowing in enough detail to delight any calculus fan.

For a subject as complex as speech synthesis, this is a clear and pleasant presentation. I learned a lot from it, and I enjoyed reading it.

The Acorn/BBC Micro, An Expert Guide

by Mike James
Published by Prentice-Hall, Inc.
$14.95, 158 pages
Reviewed by Morton F. Kaplon

Who better than the author to tell you what this book is all about? The first paragraph of the Preface summarizes it quite nicely.

"The subject of this book is the BBC Micro, its hardware, and its software, and it is aimed at anyone who has already started to plumb the depths of this fascinating machine. It is not an introduction to BASIC, nor does it attempt to explain the fundamental hardware that goes to make up any computer. Instead it plunges straight into the complexities and intricacies of this very special micro."

The book is divided into nine chapters of approximately equal length, entitled: The Hardware; BBC BASIC; The Machine Operating System; The Video Display; The Sound Generator; Interfacing; Introduction to Assembly Language; Assembly Language II; and Postscript. The author notes that material covered in the *BBC User Guide* is not duplicated in this volume except as required to make this book self-contained.

The BBC Micro is based on the 6502 microprocessor operating at a clock speed of 2 MHz with a maximum of 32K of RAM. The standard system has 32K of ROM in the form of two 16K chips. One contains the operating system (MOS) and 1K used principally for memory-mapped I/O devices. The second chip contains the BBC BASIC interpreter and the 6502 assembler; this ROM can be replaced by any of three alternative ROMs under software control. Storage is by tape cassette.

The book does as it says. It deals with this machine at the level stated and with those assumptions. If you own a BBC Micro and want to know more about it, then this book may prove useful. If you are not a BBC Micro owner, then it is difficult to see what this book may hold for you or how it could be useful to you. This is not written as a "cook book," and the implementation of the concepts presented will require serious dedication on the part of the user.

New Books

Personal Pascal

David E. Cortesi and George W. Cherry
Reston Publishing Co., Inc.
Reston, VA 1984
420 pages

This book is the wedding of George Cherry's *Pascal Programming Structures* (Reston, 1981) with the IBM PC. Our own resident intern, Reverend D. E. Cortesi, officiates.

Microprogrammer's Market 1984

Marshall Hamilton
TAB BOOKS Inc.
Blue Ridge Summit, PA, 1984
$13.50, 229 pages

You can find interesting information in this book even if you're not looking for a job. For example: prior to this year, Data General published fewer than 50 programs (ever), but the company plans to sell 200 in 1984. Yeah, Data General. The computer company with soul.

The Best of CP/M Software

John D. Halamka
SYBEX Inc.
Berkeley, CA 1984
$14.95, 252 pages

One man's opinion about 45 software packages. Although no reader will agree with all of his choices, Halamka has much software reviewing experience and writes well. One weakness, at least from a programmer's viewpoint, is the weak treatment of languages and utilities.

Macintosh! COMPLETE

Doug Clapp
Softalk BOOKS
North Hollywood, CA 1984
$19.95, 329 pages

As of this writing, one of two decent books on the Mac.

The Apple Macintosh Book

Cary Lu
Microsoft Press
Bellevue, Washington, 1984
$18.95, 383 pages

As of this writing, one of two decent

books on the Mac. This is the technical one.

The Small-C Handbook
James E. Hendrix
Reston Publishing Co., Inc.
Reston, VA 1984
$14.95, 256 pages
DDJ published Ron Cain's small compiler for a subset of the C language in the May 1980 issue. By the time (December 1982 - January 1983) *DDJ* published Jim Hendrix's greatly-enhanced Small-C, the issues containing Cain's original compiler and subsequent issues containing Small-C material had sold out. The Hendrix issues soon sold out as well. This book describes the latest version of the compiler, Small-C Version 2.1.

A Programmer's Notebook:
Utilities for CP/M-80
David E. Cortesi
Reston Publishing Co., Inc.
Reston, VA 1983
$16.95, 368 pages
You won't see this book reviewed here because of our policy of not reviewing our editors' books. We will, however, read it. Cortesi believes that programming is neither an art nor a science, but a craft. This book is about learning the craft.

The Enclopedia of Microcomputer Terminology
A Sourcebook for Business and Professional People
Linda Gail Christie and John Christie
Prentice-Hall
Englewood Cliffs, NJ, 1984
$9.95,
We've needed a good dictionary/encyclopedia of microcomputer terminology for a long time. We still do.

Apple Software Directory
Yellow Pages to the World of MicroComputers
PC Telemart/Vanloves
PC Telemart, Inc./Vanloves
Fairfax, VA, 1984
$24.95, 965 pages
The publisher bills this book as the yellow pages to the world of microcomputers, and it does have the style and heft of a metropolitan phone directory. For this third edition, its compilers have added a mediocre glossary and a list of bulletin boards, and have updated its information on software for the Apple II–III family. Scads of software.

DDJ

(Continued from page 116)

could *not* be 2 and 3."

"Mm hm. If the guy has told S that the sum was 5, S would know right away what the two numbers were. And the sum couldn't be 6, either, come to think of it, because the two possibilities would be either 3 and 3, which are eliminated because they're the same, or 2 and 4, which would mean again that he knew the numbers. Tricky."

Scrivener starts another pot brewing. He begins sniffing again.

Smith notices that the glowing end of her cigarette is almost burning the table. She picks it up and jams it in her mouth.

Scrivener again waves his hands around. "I have a puzzle that I don't think can be solved by computer."

Levin slides the pot away before the water has stopped dripping, holding his cup in place to catch the diminishing stream. When his cup is filled, he adroitly slides it out of the way, and slips the pot back in place to catch the last drops. "Any puzzle can be solved by computer, Grey."

"Oh. Write me a program, then, that answers this question: What is a horse?"

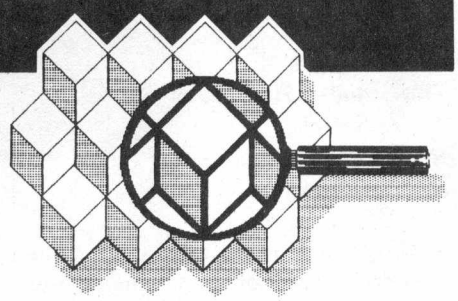
McRae giggles. "We used to make up silly questions like that at Cal, with sillier answers. Why is a duck? Because one of its feet are both the same. When is a horse? When it has a leg in each corner."

All smile, Scrivener included. "Can you describe a horse programmatically? Maybe a few FOR-NEXT loops or, even better, a little recursion?"

Well, gentle readers, can you do it? Your programs must be short and elegant. They can be in any language, although I hope you don't pick something obscure like Snobol or SPL. The algorithms may be demonstrated in a good pseudolanguage if you wish or perhaps flowcharts. The best solution to each puzzle wins a DDJ *t-shirt and will be published here. Remember that a good program is self-documenting.*

DDJ

OF INTEREST



by R. P. Sutherland

Languages

Starlight Forth Systems has a Forth compiler with macro assembler and debugging tools for the new 16-bit 65SC802 and 65SC816 microprocessors (which are plug and software compatible with the 6502 and 65C02). Options include a full assembler for all versions of the 65xxx processor family, a meta-compiler with code and high-level debugging tools, source on disk, and a full-screen editor. High-level listings of the tools for non-6502 systems are also available. For the full range of options and prices write: Starlight Forth Systems, 15247 N. 35th St., Phoenix, AZ 85032.

Wizard Systems Software has released version 1.3 of the **Wizard C Compiler**. Improvements over the previous version, which already supported the full C language, include improved syntax checking and error diagnosis of the compiler, warnings about mistaken use of the assignment operator where the equal operator was intended (mistyping = for = =), as well as indications where parentheses may be needed. Another version 1.3 enhancement is the support of the Large Memory Model, which allows Wizard C users the capability of writing programs as large as all of physical memory, without restriction. Wizard C Compiler is available for $450.00 (single CPU license) from Wizard Systems Software, Inc., 11 Willow Court, Arlington, MA 02174 (617) 641-2379.

C-Power Packs feature six libraries of functions and subassemblies to aid in debugging and to enhance code uniformity. C-Power Packs operate on Microsoft and Lattice C Compilers and include code for data base, communications, mathematics, utilities, and building *block functions*. Prices start at $99.00. For more information contact Dr. Ramal Murali, Software Horizons, 165 Bedford Street, Burlington, MA 01803 (617) 273-4711.

Macintosh

MacTools is a low-cost disk utility package that integrates many standard disk functions into one program. In addition to common disk manipulations such as copying, renaming, and deleting files and copying, renaming, and formatting disks, MacTools provides features such as: (1) copy protect/unprotect; (2) verify a disk; (3) lock/unlock files; and (4) make files visible/invisible. MacTools is available for $39.95 from Central Point Software, Inc., 9700 SW Capitol Hwy, Suite 100, Portland, OR 97219 (503) 244-5782.

SoftTech Microsystems is shipping three packages for Macintosh programmers. **UCSD Pascal Development System** allows access to mouse, graphics, and text fonts provided by Mac ROM routines for $195.00. **Fortran-77 Development System** provides an ANSI-77 subset Fortran for $295.00. The **Advanced Development Tool Kit** includes source code for graphics/mouse interface, a symbolic debugger, 68000 assembler, and a linker. The cost is $150.00. The company claims that UCSD Pascal and Apple Pascal are sufficiently compatible so as to allow porting of Apple II and III programs to the Macintosh. In addition, the company says that UCSD applications running on the IBM PC can be easily ported. SoftTech Microsystems is located at 16885 West Bernardo Drive, San Diego, CA 92127 (619) 451-1230.

The Programmer's Shop has established the **Macintosh Developer's Exchange**, a bulletin board to provide a forum for discussions related to Macintosh development. They hope to facilitate a freer flow of information and to promote the exchange of small utilities of the type usually written for in-house use. Some intriguing products are in the works that will help convert assembly language programs written for the IBM PC, the 8088 CP/M environment, and Apple IIe to the Macintosh. To participate in the Macintosh Developer's Exchange contact the Programmer's Shop, 128 Rockland St., Hanover, MA 02339 (800) 421-8006 or use the bulletin board (P-line) at (617) 826-4086.

Expert Systems

Expert-Ease is an expert system generator that runs on a microcomputer. The product is based on the artificial intelligence "inference engine" developed by Intelligent Terminals, Ltd., in Edinburgh, Scotland. An expert gives examples to the system, and once Expert-Ease has enough examples, it figures out the rules that lead to the expert's conclusions. Then the program coaches the expert to enter questions, which later will aid the nonexpert in providing information for solving new problems. Expert-Ease runs on the UCSD-p system on the the IBM PC and compatibles. Separate versions are available for DEC Rainbow and Victor 9000 (or Sirius). The full software package costs $2,000.00. A demonstration diskette and manual are available for $125.00. Contact Expert-Ease Inc., 206 Fifth Avenue, New York, NY 10010 (212) 684-4331.

The Forth Interest Group (FIG) invites you to attend the **Forth Modification Laboratory** (FORML) conference at the Asilomar conference grounds in

Pacific Grove, California, from November 23 – 25. Expert systems and artificial intelligence are among the topics that will be covered. For more information call the FIG hot line (415) 962-8653.

Jack Park's **Expert-2** (see *DDJ*, April 1984) is now optimized for use with MMSFORTH versions 2.0 and up. MMSFORTH licensees can add Expert-2 to their system for $69.95. The MMSFORTH system disk is available for TRS-80 Models 1, 3, 4, and 4P for $129.95.

Disk Storage

Chipmunk is a 3.5-inch portable disk drive for the TRS-80 Model 100 and other lap computers. The Chipmunk is powered by internal rechargeable nickel cadmium batteries or an AC adaptor. The unit plugs into the computer's expansion bus using a 1-inch wide ribbon cable and EZ-Snap connector. The Chipmunk Disk Operating System, CDOS, is contained on a ROM chip. The price is $499.00. For more information write Holmes Engineering, 5175 Greenpine Drive, Salt Lake City, UT 84123 (CompuServe 71675,527).

Miscellany

Free-Access On-Line Software Library

A Florida firm specializing in computer-readable data bases has opened an electronic software locator service to assist software shoppers in finding software products. Anyone with a telephone and a modem-equipped computer (set for 8 bits, 1 stop bit, no parity) can log on by calling (305) 845-6466. I found useful categories and thousands of software products with descriptions and prices. Shoppers can enter information, comments, or requests directly into a software vendor's electronic mail box. For more information, direct inquiries to Searchmart Corporation, 745 U.S. Highway One, North Palm Beach, FL 33408 or call (305) 845-2996.

Free Apple Software

Send one dollar to Computer Learning Center, P.O. Box 45202, Tacoma, WA 98445 for your catalog of public domain software for Apple II+ (and some Apple IIe). *Forbidden Fruit* contains thousands of programs of every possible variety, which you can order for the price of the floppy ($4.00).

CP/M Utilities Catalog

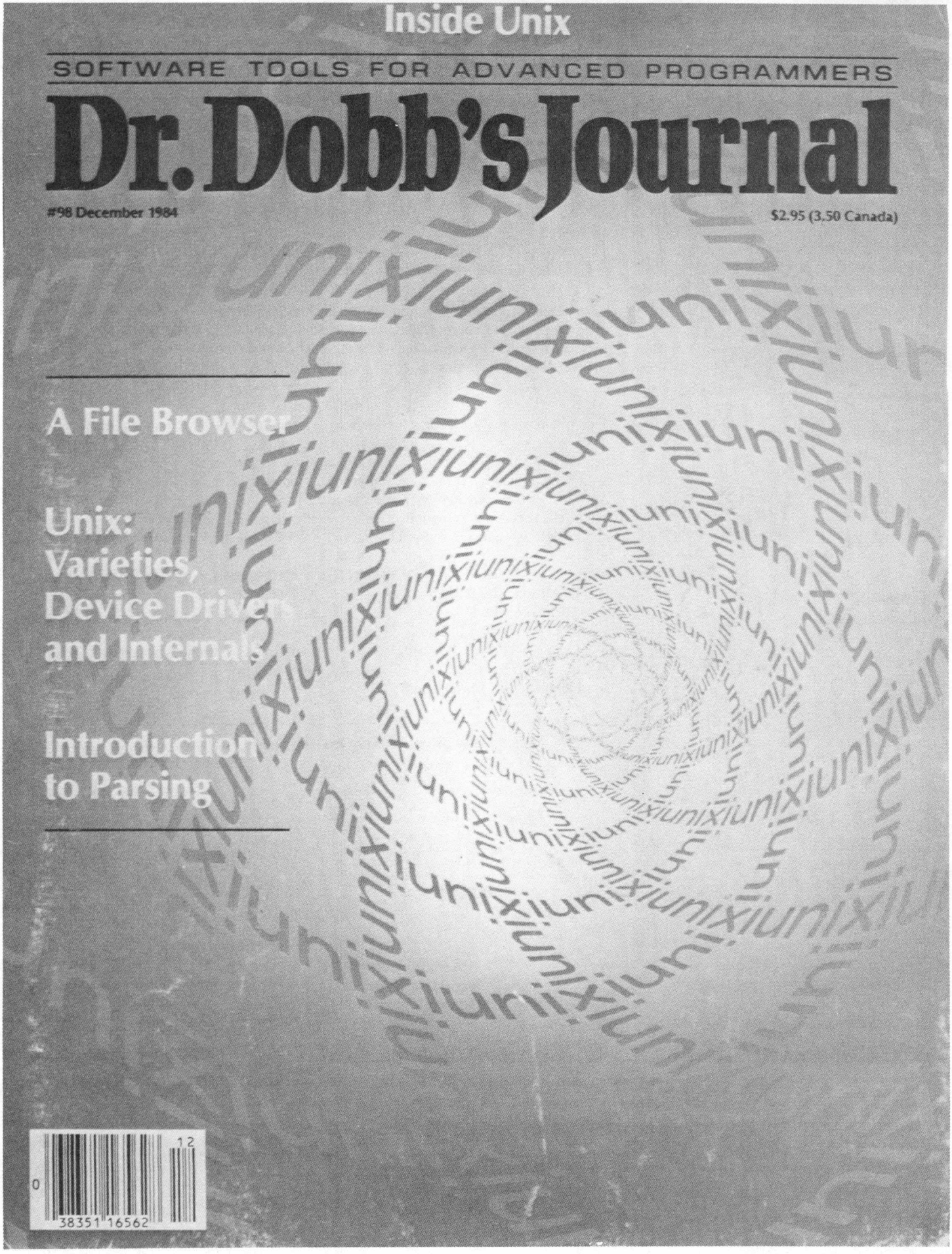
John Donohue has compiled an interesting catalog of low-cost CP/M utilities. A sample entry called "Later, listed for $22.00, allows one to accumulate lists of commands that can be postponed. According to the catalog: "When you go to lunch, type SUBMIT NOW, and they all get done. Let the computer remember what it has to do, and do it when you're not around. Obvious." None of the utilities are over $50.00 and some are as low as $7.00. Order your copy from Donohue & Co. Computer Services, P.O. Box 255, Hannibal, NY 13074.

IEEE Call for Papers

The deadline for submitting papers to the IEEE Computer Society Conference on Computer Vision and Pattern Recognition is January 7, 1985. For a copy of the call for papers with a complete list of conference topics and full details on submission of papers, write or call: Computer Vision and Pattern Recognition, P.O. Box 639, Silver Spring, MD 20901 (301) 589-8142. The conference will be held in San Francisco at the Cathedral Hill Hotel from June 9 – 13, 1985.

DDJ

Tera-drive is a high-capacity floppy disk subsystem for Apple IIe and II+. Using cobalt-coated, high-density, 5¼-inch floppy disks, this unit provides one megabyte of storage per disk. The proprietary operating system supports UCSD-p, CP/M, and DOS. The 1-Mbyte drive retails for $995.00 and the 2-Mbyte dual drive retails for $1595.00. For further information contact Eicon Research, Inc., 520 Fifth Avenue PH, New York, NY 10036 (212) 719-5353.


Inside Unix
SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal
#98 December 1984
$2.95 (3.50 Canada)
A File Browser
Unix:
Varieties,
Device Drivers
and Internals
Introduction
to Parsing
0 38351 16562 12

Dr. Dobb's Journal

Editorial

Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*
Editorial Intern *Mark Johnson*

Production

Design/Production Director *Detta Penna*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover *Scott Kim*

Advertising

Advertising Director *Stephen Friedman*
Advertising Sales *Walter Andrzejewski, Shawn Horst*
Advertising Coordinators *Alison Milne, Lisa Boudreau*
Advertising Sales *Beth Dudas*

Circulation

Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.

Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Entire contents copyright © 1984 by M&T Publishing, Inc. unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

December 1984
Volume 9, Issue 12

CONTENTS

Masthead Changes

This special Unix issue was conceived by Reynold Wiggins but he was absent from the delivery room. Midterm, Reynold removed his editor's visor and threw away the nub of his red pencil to become a CAD programmer at Fairchild. So yours truly, Randy Sutherland, assisted the Doctor on this baby. Reynold Wiggins has been involved with *Dr. Dobb's Journal* almost since the beginning and we expect an ongoing relationship. When it comes to programming on microcomputers, it is hard to match his zeal!

New names will appear on the masthead next month: Frank DeRose, Assistant Editor; and Alex Ragen, Technical Editor.

Next Year

Next month we feature an article on how to "Fatten Your Mac" for a lot less than Apple's price. Also, we will review Logitech's Modula 2 compiler and STSC's APL interpreter. February is the Gala Anniversary Issue, 100 months of *DDJ*! In March we focus on artificial intelligence for microcomputers and announce the winner of the AI competition. April will feature human interface design and May will feature graphics algorithms. June will be the special telecommunications issue. Please submit manuscripts for the telecommunications issue by the end of February.

This Month's Referees

Dennis Allison, Stanford U.
S. M. Bellovin, AT&T
Wayne Chin, Hewlett-Packard
Bob Desinger, Hewlett-Packard
Mohammed El Lozy, Harvard U.
Jim Fleming, Unir Corporation
Angel Gomez, Telecomp, Inc.
John Keyes, Microsoft
Ben Laws, North Texas State U.
John Rogers, Fortune Systems
Joseph Sharp, Micro Science Associates

STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION

(Act of August 12, 1970, Section 3685, Title 39, United States Code)

1. Title of Publication: Dr. Dobb's Journal, Publication No. 307690.
2. Date of Filing: October 12, 1984.
3. Frequency of Issue: Monthly (12 issues, $25).
4. Location of Known Office of Publication: 2464 Embarcadero Way, Palo Alto, CA 94303.
5. Location of Headquarters of General Business Offices of the Publishers: 2464 Embarcadero Way, Palo Alto, CA 94303.
6. Names and Addresses of Publisher, Editor, and Managing Editor: Publisher, Laird Foshay, 2464 Embarcadero Way, Palo Alto, CA 94303. Editor, Michael Swaine, 2464 Embarcadero Way, Palo Alto, CA 94303. Managing Editor, Randy Sutherland, 2464 Embarcadero Way, Palo Alto, CA 94303.
7. Owner: Otmar Weber, 2464 Embarcadero Way, Palo Alto, CA 94303; and C. F. Von Quadt, 2464 Embarcadero Way, Palo Alto, CA 94303.
8. Known Bondholders, Mortgagees, and Other Security Holders Owning or Holding 1 Percent or More of Total Amount of Bonds, Mortgages or Other Securities: None.
9. Extent and Nature of Circulation:
   A. Total number of copies printed: Average number of copies each issue during the preceding 12 months: 31,407. Actual number of copies of single issue published nearest to filing date: 35,550.
   B. Paid Circulation. 1. Sales through dealers and carriers, street vendors, and counter sales. Average number of copies each issue during preceding 12 months: 6,099. Actual number of copies of single issue published nearest to filing date: 6,625. 2. Mail subscriptions. Average number of copies each issue during preceding 12 months: 18,268. Actual number of copies of single issue published nearest to filing data: 20,718.
   C. Total Paid Circulation. Average number of copies each issue during preceding 12 months: 24,367. Actual number of copies of single issue published nearest to filing date: 27,343.
   D. Free distribution by mail, carrier, or other means, samples, complimentary, and other free copies. Average number of copies each issue during preceding 12 months: 230. Actual number of copies of single issue published nearest to filing data: 587.
   E. Total distribution. Average number of copies each issue during preceding 12 months: 24,597. Actual number of copies of single issue published nearest to filing data: 27,930.
   F. Copies not distributed. 1. Office use, left over, unaccounted, spoiled after printing. Average number of copies each issue during preceding 12 months: 1,820. Actual number of copies of single issue published nearest to filing date: 2,200. Returns from news agents. Average number of copies each issue during preceding 12 months: 4,990. Actual number copies of single issue published of nearest to filing date: 5,420.
   G. Total. Average number of copies each issue during preceding 12 months: 31,407. Actual number of copies of single issue published nearest to filing date: 35,550.

I certify that the statements made by me above are correct and complete.

Laird Foshay, Publisher

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

Varieties of Unix *by Alan Walworth* **24** A comparative overview of Unixes for microcomputers with a brief history of Unix and comments on its future, plus a guide to choosing a Unix

Unix Device Drivers *by John Bass* **38** Unix Version 7 drivers are the point of departure for this inside look at the Unix I/O subsystem and Unix device drivers

A Unix Internals Bibliography *by John Rogers* **50** An expert's guide to internals documentation so you won't have to "grep for it"

A File Browser Program *by John Johnson* **60** For those times when all you want to do is look through a file

An Introduction to Parsing *by Henry Seymour* **78** How to implement parsing schemes for assemblers, editors, or adventure games

COLUMNS

Dr. Dobb's Clinic *by D. E. Cortesi* **16** Around and around with Shugarts and Tandons **(Reader**

CP/M Exchange *by Bob Blum* **20** The early days of CP/M; Using large sectors

16 Bit Software Toolbox *by Ray Duncan* **88** Sneak 80286 Preview

C/Unix Programmer's Notebook *by Anthony Skjellum* **96** Long pointer corrections, programming philosophy

Computer Calisthenics *by Michael Wiesenberg* **120** Puzzles

DEPARTMENTS

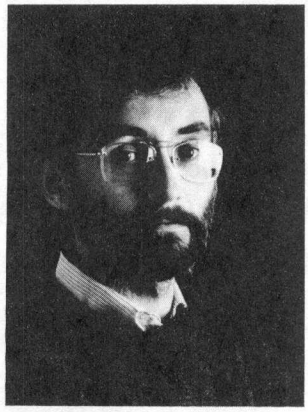
Editorial **6**

Letters **8**

Software Reviews **106** NCI Coherent, Turbo Pascal Version 2.0 (MSDOS and CP/M)

Of Interest *by R. P. Sutherland* **124**

EDITORIAL



Arguments for not doing a special issue on Unix: 1) I flew to New York on October 15 to attend a Unix Expo; I was not alone in finding the show slow and boring. (Well, not entirely boring; I did see some C programming tools that were of some interest, Jim Joyce's seminars seemed well-attended, and one company was showing off an intriguing product that implements Smalltalklike object-oriented programming in a Unix environment, but the show attendance was sub-moblevel.) 2) I drove up the peninsula to San Francisco on October 26–27 for the PC Faire; the Faire did nothing to reassure me that Unix is on the verge of taking over the exciting world of IBM PCs and compatibles. (OK, I did see some friendlifying frontends for PC Unixes.) 3) Dr. Dobb's readers have read all the inflated claims about Unix market share, don't believe them, and anyway don't read *DDJ* for that sort of thing; as one irascible newsletter editor who pontificates under the name Felgercarb N. Eloi recently wrote, "it was absurd to assert that Unix was or ever would be the best operating system for the dentists and candle-makers who comprise the mass personal computer market." (But many of our readers do use Unix.)

Argument for doing a special Unix issue: our readers are not dentists or candlemakers.

This special issue on Unix is a new departure for *DDJ*. We have in the past done special issues on Forth and telecommunications, and current plans call for us to continue doing these special issues. We will in fact be doing more special issues in the future. But we're changing our approach to them: instead of devoting an entire month of the magazine to a topic, we are now shining a more focused light on the topic at hand, leaving room for other articles in the same issue. The idea is that we're consciously structuring these special sections to deal in some depth with a relatively narrow slice of an interesting topic.

Consider the current issue's special section, which we have named Inside Unix. Here's what we think we have done here; you can let us know how well we succeeded. The Unix section includes only three articles (plus the regular "C/Unix" column and attention to Unix in "Of Interest"), and it centers on the Unix drivers article by John Bass.

The Bass article is directed at people who want something more than marketing hype on how many Unix licenses have been sold, or analysis of where Unix fits into the office of tomorrow. It takes you deeper into the Unix system, hence the section title Inside Unix.

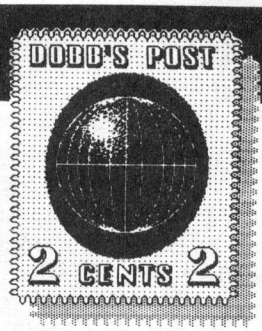
Rogers' richly annotated bibliography of Unix internals is again not for the tyro but is a useful tool for the programmer with a need to know about Unix internals. We hope to supply such bibliographies whenever they are appropriate. The overview article by Walworth is here to provide a background for the other pieces and to clear up some confusion about versions of Unix for those who haven't got it all quite straight.

You're right if you're thinking that all this is some kind of experiment. As Castle, the slanderous stereotype of a humanities professor in B.F. Skinner's *Walden II*, said, the experimental attitude has the wonderful feature of letting one be utterly confident and self-righteous without knowing anything. We're experimenting, cautiously, with the magazine, and once we see the results of our present experiments, we'll probably continue experimenting.

Michael Swaine

Michael Swaine

LETTERS



Forum

Dear *DDJ*,
I was interested to see Alex Cameron's contribution of standard (K&R) fopen-/fclose functions for BDSC in the August '84 *Dr. Dobb's Journal* ("BDSC Runtime Solution," page 118). I contributed code with a similar goal to the C User's Group over two years ago. Cameron took the conventional approach of using alloc/free to allocate buffer space in the "heap," the region of memory between the top of the externals and the bottom of the stack. I was more concerned to avoid the tendency to allocate too many buffers and bash the stack, so I #defined the maximum number of buffers (simultaneously open files) as NIOBUFS before main( ), and reserved the buffers in the external data space. Thus, inadequate memory problems would be obvious at link time. My code was set up to be #included at three points in main( ). I have recently ported eleven substantial programs from BDSC to Computer Innovations C86 and found the use of standard fopen/fclose to be a real help.

More recently, along with a text formatter which I wrote, I contributed a library of 43 general purpose functions to the C User's Group ("Martz Library Disk"). Your readers writing in C may save some time by getting a copy of these. Most of the functions are for character string manipulation. For example, argmatch( ) finds arguments to main( ) in any sequence, with leading/trailing ambiguities, optionally deleting them from the argc, argv list once found. fbrkout( ) breaks a long string into pieces of specified width and hands it to an output stream. Breaks are between words. It has options for indentation and for representing non-printable characters. badname( ) verifies that a filename is valid according to CP/M file-naming rules; if not, it issues a detailed error message. findwords( ) counts the number of words in a string and sets up an array of pointers pointing to each. pack( ) packs strings into a big buffer for later retrieval, returning a pointer to the beginning of each. substitute( ) replaces all instances of a specified string with a specified replacement. todelim( ) finds the first instance of a specified delimiter (which can be one or more characters) and splits the original string into left and right portions excluding the delimiter. ynqd( ) is one of the most useful: "Yes No Question with Default." As an example of its use, to ask whether the user needs help:

```
if (ynqd("Do you need help", YES))
    givehelp( );
```

This displays on the console:

```
Do you need help? (y/n)
  (default = YES)
```

Simply hitting a carriage return [signals] the default YES .... In addition to my own modest contributions, many others have contributed tools and programs totaling about forty 8-inch SSSD CP/M disks. Readers programming in C should avail themselves of these by writing to the C User's Group, 112 N. Main, Box 287, Yates Center, KS 66783; phone 316-625-3553. Six issues of the CUG Newsletter are $10, and 8-inch SSSD CP/M disks are only $8. Several 5-inch disk formats are also available (Apple II, Heath/Zenith, TRS-80, Northstar, Osborne, and others).

Sincerely,
Eric Martz, Ph.D.
48 Hunter's Hill Circle
Amherst, MA 01002

Dear *DDJ*,
I read the article by Joe Barnhart about the fast Fourier transform (FFT) in the September 1984 issue. A disadvantage of the FFT is that all the numbers in the transform are complex, requiring complex arithmetic. A complex addition requires two additions of real numbers, and a complex multiplication requires four real multiplications and two real additions. The user has to use twice as much memory to store a complex array of numbers for the FFT. I know of variations of the FFT algorithm that repack the real input numbers into a complex array half the original length. The FFT is performed and the result unpacked. The FFT of the repacked array uses less operations. However, the packing and unpacking algorithms are complicated.

An alternative to the FFT is the fast Hartley transform (FHT). The Hartley transform is similar to the Fourier transform, except that the input and output numbers remain real. Since most applications of the Fourier transform usually involve real input data, the FHT is more suitable for the average user. Since the FHT does not involve any complex operations, the FHT requires fewer multiplications and additions than the FFT algorithm presented in the article. The basic equation for N-point Fourier transform is

$$Xf(k) = \sum_{n=0}^{N-1} x(n)\,(\cos((2\pi / N)\,nk) - i\sin((2\pi / N)\,nk))$$

where $x(n)$ is the complex input, $Xf(k)$ is the Fourier transform, and i is the square root of -1. The equation for the N-point Hartley transform is

$$Xh(k) = \sum_{n=0}^{N-1} x(n)\,(\cos((2\pi / N)\,nk) + \sin((2\pi / N)\,nk))$$

where $x(n)$ is the real input and $Xh(k)$ is the Hartley transform. Note that the

Hartley transform does not have any complex arithmetic involved. The inverse Fourier transform is different from the forward Fourier transform. The inverse Hartley transform is the same as the forward Hartley transform. The FFT makes use of the symmetry of the Fourier transform, which leads to the butterfly graph shown in the article. The Hartley transform also has a symmetry which leads to a FHT algorithm that uses a double butterfly graph. The user can switch the results from a Hartley transform to the Fourier transform through the equations

$$\text{REAL}[X_f(k)] = [X_h(k) + X_h(-k)]/2$$

$$\text{IMAG}[X_f(k)] = [X_h(k) - X_h(-k)]/2$$

Enclosed [see Listing One on page 12] is a C listing of a subroutine that performs a FHT. Many common operations of the frequency domain are computed faster in the Hartley domain. For example, convolution and cross correlation require fewer operations when done with the Hartley transform.

I hope you will let your readers know about the Hartley transform, because it is better suited for small computer systems than the Fourier transform. For a good reference, see Ronald Bracewell's article, "The Fast Hartley Transform" in the August 1984 issue of *Proceedings of the IEEE*. Another reference is my article "An Algorithm for the Fast Hartley Transform" in the Stanford Exploration Project #38.

Sincerely,
Ron Ullmann
Picture Element Ltd.
635 Waverly
Palo Alto, CA 94301

Dear Doctor:
In a recent issue, a reader told you about how a patch to the DRI macroassembler RMAC, which he received from Manx, solved his need to use underscores and periods in identifiers. Anyone mixing C or Pascal with assembly language modules linked together will find this enhancement to RMAC useful.

Because the above-mentioned patch may have wide appeal to your audience of system hackers, I am including a listing [see Listing Two on page 13] that achieves the purpose, and which can also be customized for MAC. This same listing was available from several bulletin boards, but given the huge amount of software that those boards carry, it might have been unnoticed by many.

I would like to point out that any potential commercial firm interested in this patch should make arrangements with me prior to using it.

In an entirely different matter, Ray Duncan and thereafter some readers discovered many problems with MASM, the Microsoft/IBM 8086 macroassembler. It's a sorry state of affairs that some companies can provide such low level of quality to the marketplace and still be praised by much of the press. But the reason I'm writing you about this is that I'm using two versions of MASM; one is 1.07 and the other is 2.04. The former one seems to have less problems than the 2.04. Many of the bugs found by Mr. Duncan are not present in version 1.07, which seems to perform normally in these tests. I would recommend readers to try to obtain this version and see if I'm right.

Some other bugs are still present, even in this better version, plus problems resulting from being a two-pass assembler. I don't know or have RASM, but ASM-86, which comes with CP/M86, is a three-pass assembler, which allows it to do a better allocation of space before actually starting the assembly. This eliminates many nonsense NOPs that MASM needs to scatter through the object code, degrading even more a slow-performing chip like the 8088.

Regards,
George Blat
Blat Research +
Development Corp.
8016 188th Street SW
Edmonds, WA 98020

Dear *DDJ*,
"Designing a File Encryption System" (*DDJ*, August 1984) by Thomas and Thersites is delightful. Can you imagine the hilarity in the halls at the Puzzle Palace when their cafeteria serves recycled permutation table generators? I can just see the FBI on stakeout at the embassy supermarkets watching for a surge in sales of one pound bags of leguminous encryption aids.

Sincerely,
Adam Fritz
133 Main Street
Afton, New York 13730

DRI Support

Dear Editor:
I am writing in response to Steve Conley's letter in the August issue of *Dr. Dobb's Journal*. Mr. Conley described a bug he found in RMAC. He was upset when people at Digital Research told him that they didn't know when, or if, the problem would be fixed. I would like to address several of the concerns which he raised.

It has always been Digital Research's policy to be honest with our customers. I truly wish we could fix all the bugs in all of our products as fast as we would like. Unfortunately, the reality is that this doesn't happen. Instead we try to prioritize problems and do all we can to make sure our customers are aware of them. Fixing this problem has a low priority because it is possible to work around the problem and because RMAC is an 8-bit product. If we had promised Mr. Conley that the problem would be fixed "soon" he probably would have been satisfied. I firmly believe, however, that it is much better to be honest. Therefore the engineer described the status of the problem to Mr. Conley and discussed with him two possible work arounds for the problem. To me this represents quality support rather than indifference.

Providing quality technical support to a large customer base can be difficult. The method which we have chosen to provide this support is our Professional Programmer Support Program (PPS). PPS includes unlimited toll-free phone access to our engineers, a technical newsletter and a subscription to CompuServe so that the customer can access our data bases. This package is available for only $250 per year for each customer contact person.

I think that PPS is what Mr. Conley was referring to when he mentioned a $250 software maintenance package. Maintenance, however, is available to all registered users of our products. We use the registered users data base to notify customers whenever a new re-

lease is available. Unfortunately we could not find Mr. Conley at the address he listed as either a subscriber to PPS or as a registered user. Because of this, we will not be able to notify him when updates occur for any of his DRI products.

Mr. Conley mentioned problems he has had reaching DRI by phone. Recently we have made some improvements so that it is now relatively easy to reach our Warranty Support Department. They do not provide customers with technical support but do provide customers with warranty support. Their number is (408) 646-6464.

In summary, I am very proud of the support we, at DRI, provide our customers. I believe we have found a very good way to provide a much needed service to our customers.

Sincerely,
Marion Brown
Support Center Manager

160 Central Avenue
Pacific Grove, CA 93950

Caveat Emptor

Dear *DDJ*,

The purpose of this letter is to warn *Dr. Dobb's* readers of problems in dealing with JRT.

Early in June of this year I called JRT on the phone and ordered a copy of their Modula-2. I was told that the product would be shipped within a week. On 20 June my VISA account was billed $102.95 for Modula-2 plus shipping.

When the product had not arrived by August, I tried over a period of three weeks, repeatedly and at all hours, to reach JRT at their listed phone. The phone was never answered. Finally, at the end of August I sent JRT a certified letter, return receipt requested, describing the situation and asking that either my copy of Modula-2 be shipped or that my VISA account be credited. The return receipt came back to me on 7 September, signed by Jim Tyson of JRT.

It is now 24 September and I have heard nothing from JRT, I have not received Modula, and my VISA account has not been credited. While most of my experience ordering from computer suppliers has been extremely satisfactory, I think JRT's actions, taking your money and not shipping the product, are at best a sharp business practice and, at worst, border on fraud.

Sincerely,
R. A. Langevin
7621 Fontaine Street
Potomac, MD 20854

DDJ

Letters (Text begins on page 8)

Listing One

Fast Hartley Transform

```
/* The FHT performs the fast Hartley transform over an an array of
 * floating point numbers.  The array of real numbers are pointed to by
 * FX.  LENGTH is the number of points in the array and must be a power of
 * two (i.e. 16, 32, 2048).  The Hartley transform of the array is stored
 * in the array FX upon completion.  If LENGTH is not a power of two, FHT
 * returns a value of -1.  Otherwise, FHT returns 0.  By Ron Ullmann
 */
(length, fx)
  int length;  double *fx;
{
        int             ii, kk, jj, ll, istep;
        double          *pba, *pbb, *pbc, *pbd;
        double          temp1, temp2, arg, fcos, fsin, dsin, dcos;

        /* Test to see if length is a power of two and is not zero.  */
        for (kk = length; (kk & 1) == 0; kk >>= 1) ;
        if (kk != 1)  return (-1);
        /* Reorder the data */
        arg = sqrt (1. / (double) length);          /* scale the data */
        jj = 0;
        for (ii = 0, kk = 0; ii < length; ++ii, kk += jj)
          {     if (ii <= kk)
                  {     temp1 = *(fx + kk) * arg;
                        *(fx + kk) = *(fx + ii) * arg;
                        *(fx + ii) = temp1;
                  }
                for (jj = length>>1; kk >= jj && jj >= 1; kk -= jj, jj >>= 1) ;
          }
        arg = 3.14159265358979323846264338;
        for (jj = 1; jj < length; jj = istep)
          {     istep = jj << 1;
                dcos = cos (arg);  dsin = sin (arg);
                arg /= 2.;
                for (ii = 0; ii < length; ii += istep)
                  {     pba = fx + ii;  pbc = pba + jj;
                        temp1 = *pbc;
                        *pbc = *pba - temp1;  *pba += temp1;
```

(Continued on next page)

Letters (Listing Continued, text begins on page 8)

Listing One

```
                }
            fcos = dcos;  fsin = dsin;
            for (ll = 1, kk = jj - 2; ll < (jj >> 1); ++ll, kk -= 2)
                {       for (ii = ll; ii < length; ii += istep)
                        {       pba = fx + ii;  pbb = pba + kk;
                                pbc = pba + jj;  pbd = pbc + kk;
                                temp1 = fcos * *pbc + fsin * *pbd;
                                temp2 = fsin * *pbc - fcos * *pbd;
                                *pbc = *pba - temp1;  *pbd = *pbb - temp2;
                                *pba += temp1;  *pbb += temp2;
                        }
                        temp1 = fcos*dcos - fsin*dsin;
                        fsin = fsin*dcos + fcos*dsin;  fcos = temp1;
                }
            if (jj > 1)
                        for (ii = (jj >> 1), ii < length; ii += istep)
                        {       pba = fx + ii;  pbc = pba + jj;
                                temp1 = *pbc;  *pbc = *pba - temp1;
                                *pba += temp1;
                        }
        }
    return (0);
}                       /* end of FHT */
```

End Listing One

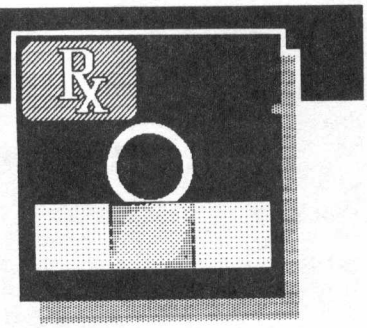
Listing Two

```
;*****************************************************************
;
;                   Patches for MAC and RMAC
;                   ------------------------
;
;                        by George Blat
;               Blat, Research + Development Corp.
;                         8016 188th SW
;                       Edmonds, WA 98020
;
;
;*****************************************************************
;
;
;The following changes are (c)1983 Blat R+D Corp. Permission is
;granted to use these patches only in non-commercial applications.
;MAC and RMAC are trademarks of Digital Research, Inc. which holds
;ownership and all rights to the original programs.
;
;*****************************************************************
;
;
;Mac and Rmac are two reliable assemblers developed by Digital
;Research which have a good number of useful features. It seems
;natural to get the most out of them.
;
;Among the features that can be added to Mac and Rmac, are the
;ability to use the period '.' and the underscore '_' as part of
;symbol names such as labels, even as first character of the
;symbol. The underscore, for instance, makes a much better word
;separator than the dollar '$' sign when used in a multi-word
;label. In a dense program listing, it's certainly easier to find
;STAT_PORT than STAT$PORT, and @hl_to_de than @hl$to$de.
;
;By the same token, I don't agree with the decision of Digital
;Research of making the dollar sign a don't care character. It
;introduces confusion as it allows symbols that don't look the
;same to be equivalent.
;
;In addition, RMAC can be easily patched to create .REL files
;where the global (external) names have up to 7 active characters.
;This helps by allowing you to create more meaningful symbol names
;and therefore improve program legibility. This change is still
;entirely compatible with the industry standard Microsoft format.
;
;The following patches should be assembled with MAC (not RMAC)
;and the resulting hex file should be applied over the original
;programs with DDT, SID or ZSID. KEEP AN ARCHIVE COPY OF THE
;ORIGINAL MAC OR RMAC BEFORE PATCHING.
```

```
false   equ     Ø
true    equ     not false

rmac    equ     true            ;select one and only one of these
mac     equ     false           ;true

        if      rmac
global7         equ     true    ;set to false if you don't want
                                ;7 char globals
patcharea       equ     13bh
dollarcounts    equ     1d7ah   ;set this to false if you like to
                                ;keep the dollar as a don't care char
checkalfa       equ     1d9ch
toup            equ     2844h
        endif

        if      mac
copyrite        equ     1Ø3h    ;shorten but keep the copyright notice
dollarcounts    equ     1834h
checkalfa       equ     1853h
        endif

        if      mac

        org     copyrite
        db      '(c)1977 DRI'
patcharea:
        endif

        if      rmac
        org     patcharea
        endif

        cpi     '.'
        rz
        cpi     '_'
        rz
        cpi     '?'
        rz
        cpi     '@'
        rz
        if      rmac
        call    toup
        endif
        sui     'A'
        cpi     'Z'-'A'+1
        cmc
        ret

        if      rmac and global7

compare equ     12d6h
setit7  equ     12dbh

        org     compare
        cpi     8               ;replaces cpi 7
        org     setit7
        mvi     a,7             ;replaces mvi a,6

        endif

        if      dollarcounts
        org     dollarcounts
        nop                     ;replaces mov m,a
        endif

        org     checkalfa
        call    patcharea       ;replaces cpi 3f
        cmc                     ;jz     ldbl

        sbb     a               ;cpi    4Ø
        ret                     ;jz     ldbl,   etc.


        end
```

End Listing Two

DR. DOBB'S CLINIC



by D. E. Cortesi, Resident Intern

The Intern Loses His Head: A Cautionary Tale

It was a dark and stormy night. Rain lashed against the windows of the Clinic. The Intern sprawled across a gurney. He pushed his mask aside and slurped from a paper cup of coffee. Winced. Sighed. And spoke, his voice barely audible above the wind. "I lost a patient," he said. "It was my own damn fault, sheer arrogance. But I paid for it . . . ."

The New Drives

I had a pair of Shugart 800s (he went on), single-sided 8-inch drives. They worked fine, but I wanted more. Six hundred Kb just isn't enough for some of the things I do. Double-sided drives have 1200K, and with 256 directory entries they'd hold any project short of the U.S. Census.

And they're much faster—you know me and speed. Modern drives step at 3 milliseconds, twice as fast as my old drives, and with twice the data under the heads they incur fewer and shorter seeks on the average.

But I wanted not merely new drives but half-height drives. No practical reason—just because they're sexy. Oh, a pair of half-heights is a lot less bulky than two normal drives, but my real motive was . . . aesthetic.

So I placed an order with Floppy Disk Services (741 Alexander Road, Princeton, NJ 08540; 800-223-0306) for an enclosure with power supply, two drives, and a cable. It cost just over $1300.

Floppy Disk Services will configure just about any combination of drives, and I had to choose a make of drive. Here was the first place I really went wrong. I asked for Shugart 860s for no better reason than that the old Shugarts had given such good service. If I'd only asked for advice—from FDS or a local guru—I wouldn't have ended up holding a detached head in my hand.

The Interface

The disks came a month later. The enclosure was handsome, the power supply looked sturdy, and the drives were . . . aesthetically delicious. Open cutouts on the back panel might have allowed air to shortcut the fan, but a piece of cardboard from the shipping carton blocked them neatly.

I knew that all drives have a zillion jumper options and that the odds against FDS plugging them right for my system were astronomical, so I'd ordered the Shugart manual. That was a good move. The manual made it glaringly obvious that floppy technology had changed quite a bit since my disk controller, a venerable CCS 2422, had been designed.

My first problem was how to control the drive motors. The old drives rotated all the time but only closed the heads onto the disk when the 1793 asserted the head-load line (clack!). When five index holes passed without any activity, the 1793 would automatically reset the head-load (clink!) so the disk could turn freely.

The new drives loaded the heads as soon as a disk was inserted. The way their options were strapped on arrival, they would rotate the disk as long as the drive-select line was asserted. My BIOS (which followed the original CCS BIOS) never cleared the drive-select port, so the last-used drive would rotate indefinitely—with the heads pressing on the disk. Not good.

The new drives allowed a jumper option that would start the motor only when the drive was selected *and* the head-load signal was asserted. Another option would keep it turning for 5 seconds after the fall of head-load then stop it. These options solved the first problem very nicely.

They also created the second problem. The new drives might take 168 milliseconds to come up to speed after head-load was asserted. On the other hand, they *might* be ready to go instantly—if the motor were still turning from the last access. How could the 1793 controller chip know when to wait for the motor to start and when it needn't wait?

Well, these drives emit a signal that was new to me. The line, True Ready, is pin 8 of the interface. The drive asserts it when the motor is up to speed and the head is stable; this is exactly what the 1793 chip needs to tell it to go ahead with a read or write. What's more, the 1793 has an input, Head-Load Timing, that takes exactly that information. Unfortunately, pin 8 on my hoary old disk controller board is a no-connect. The Head-Load Timing input to the 1793 is developed by a 50 millisecond one-shot on the board itself.

OK, I told myself, that's why I own an X-acto knife and a soldering iron, isn't it? It took most of a weekend to work out that it is possible, using only existing components of the CCS 2422 board, to implement True Ready as the Head-Load Timing signal of the 1793. And it worked. I got the Shugart 860s reading and writing with all four heads.

The Decapitation

After a few days, drive B started to fail intermittently. Sometimes it was "record not found"; sometimes it was a CRC error on a sector ID; sometimes an error on a sector proper. Strange thing, it was worse with Dysan disks than with junk disks.

OK, now I had a choice. I could refit the old drives, pack up the new ones, cart them to the UPS office, pay to ship them back to New Jersey, and wait (how many weeks?) for them to come back. Or I could take off the cover and

have a peek myself. One little peek wouldn't hurt.

With the covers off, I could *hear* the problem. The rotational speed was varying enough to be audible. Was it a bad motor? No! There was simply too much pressure on the disk jacket. The drive has a spring-loaded shoe that bears down on the disk jacket. I put out a cautious pinky and lifted the shoe: the disk sped up and a bad sector became readable. I released it: the drive slowed down audibly and an error occurred. Maybe the Dysan disks had thicker jackets or a little more internal friction.

This is the point at which I lost the patient. I looked into the drive and decided that the pressure of the shoe was set by the position of a big plastic cam. The same cam raised the head and tripped the eject arm when the drive was opened. Its position was maintained by two setscrews. They were covered with touch-me-not varnish. Should I ship the drives back, or should I get out a screwdriver and try an adjustment? You can guess what I did.

The cam adjustment was tricky. The cam worked against a heavy torsional spring that fought every move. Worse, after I started tinkering, it dawned on me that it wasn't the cam at all. The pressure shoe just floated in it, held down by a weentsy little spring of its own. I got the cam back where it should be, more or less, fixed the shoe pressure, and buttoned it up. And it worked! My goodness, was I relieved. Maybe I had flubbed a little, I told myself, but, by golly, I had diagnosed and repaired the problem.

Ah, the bliss of ignorance. That big spring was working away on the head-lift cam, gradually shifting the setscrews. The head was lifting less and less when the disk was ejected. I noticed a little extra noise when the disk popped out of the drive but thought nothing of it. Until the drive quit working.

Catastrophe!

It wouldn't read or write on side one. I took the cover off again and looked inside, and a bowling ball dropped into my stomach. The upper head was dangling on its fragile wires, completely detached from the arm!

A quick look with a dental mirror revealed the awful truth. In the Shugart 860, the upper head is suspended in a plastic frame. The sum total of its support is a pair of tiny bronze straps. They are less than a millimeter wide and no thicker than a sheet of paper. If the head doesn't lift far enough, the edge of the disk jacket will tweak it as the disk pops out. The head will flop back and forth on these flimsy metal straps. Do it often enough, the straps will fail and the head will dangle on its wires.

OK, there I was with a thoroughly inop drive and not a prayer of making a warranty claim. So I started making phone calls, trying to find a professional to fix the drive. What I got was more bad news bulletins than a stock ticker in 1927.

The first guy I talked to told me that, although Shugart 800s were "built like Mac trucks, they run forever," the 860 half-heights were "uh, let's say, a little fragile. You shoulda got the Tandon 848-2, it's a real nice drive." Thanks for the tip; what about replacing the upper head?

No, the only repair unit for the heads was the entire head assembly, upper and lower both, and it cost just under $300. Fine, I said, that's less than a whole drive, when can we get one?

That could be a problem, I was told. "We think of Shugart as being three miles and thirty days away. We might get lucky and have your part in three weeks. Probably more, though, since Shugart has given up on that drive."

They have? I called Shugart. Yes, a nice lady in Marketing told me, they were no longer selling the 860. She thought it might be because the market for 8-inch drives was so much smaller than that for 5-inch ones. No, it definitely was not because the 860 was a bad drive. They were still making the 800 model, and they'd support the 860 for several years. Repair parts? Certainly; just call our one and only official distributor in Los Angeles.

The Tandon 848

So I've got not just a busted drive but a busted *orphan* drive, see? I'm out at least three weeks and $300 for a new head assembly, then I'll have a working orphan drive. Rubbish. I called Priority One Electronics. What about the Tandon? Sure, it's a fine drive, we have 'em in stock, and they are 100% compatible. Plus they're on sale.

The new Tandon came in only two weeks, but the Tandon manual didn't; it was back ordered. The drive has at least as many jumper options as the Shugart, but (of course) they had different names silk-screened onto the board. The piece of cardboard that was stuffed into the drive's gullet to keep the heads apart during shipping had a cryptic table of jumper options. Some of the factory settings it showed didn't match the reality of the board. Others it named turned out not to be jumpers at all but traces that could be cut or soldered.

I called Priority One repeatedly; I even talked them into reading me the Tandon manual over the phone. The guy who read it to me was not the world's best interpretive reader. I got the impression that I couldn't use True Ready and control the motor with Head-Load Timing at the same time. There were too many ambiguities.

Finally, the manual arrived. With tax it cost just under a dollar a page. You know, the one really good thing about the Shugart 860 drives is the manual. It is clear and well organized. The Tandon manual isn't. Its description of the jumper options is cryptic, ambiguous, and actually wrong in a few places. No wonder I'd been confused by a hasty reading over the phone!

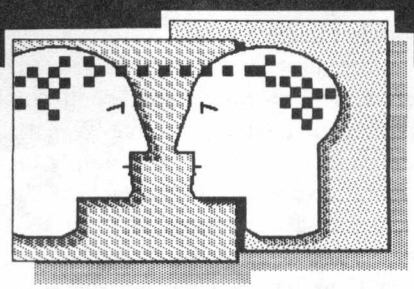
Fortunately, the drive itself is nice. Much better than the Shugart. I can't say how robust the head suspension is, since the head assembly is invisible and I have no intention of taking a screwdriver to that drive. But it's much quieter in operation. The Shugart (the remaining 860 is now drive B) makes a harsh buzz when it seeks and loud snapping noises as its door-lock solenoid operates. The Tandon seeks with an oily purr that's barely audible over the hum of a cooling fan.

The Shugart doesn't know where its head is when it powers up; until it is homed, it won't read reliably. The Tandon homes itself when power is applied, so it is ready to go on the first command.

The Tandon isn't really compatible, though. Oh, it responds to the same commands the same way—almost. But its power supply connector isn't the same, and the threaded holes for side mounting are in different places than

(Continued on page 119)

CP/M EXCHANGE



by Robert Blum

I've run a little late completing the subroutine to handle buffered disk I/O that I promised to run this month. In its place I want to talk about the early years of CP/M and how it became the most popular operating system for 8-bit computers. I also want to touch on the benefits of using disk sectors larger than 128 bytes. Both of these topics were requested by some nice folks that I recently met at a user group meeting. They thought a discussion without the bits and bytes, or at least minimizing the technical details, would be most helpful—especially for the hobbyist who is new to CP/M and isn't familiar as yet with assembly language programming.

In the Beginning

From its inception, CP/M was targeted for the 8008's successor and Intel's latest brainchild, the 8-bit 8080 microprocessor. While a software consultant for Intel, Dr. Gary Kildall had written the earliest versions of CP/M for his own experimental machine. His original development system included one of Shugart Associates first 8-inch disk drives, which had just come from equipment life testing and was about to take its last step. Fortunately, plenty of magic was left to finish CP/M. No one realized it at the time, but the combination of the 8080 CPU, CP/M, and two 8-inch disk drives was soon to become the standard of the then fledgling personal computer industry.

Within a few years, the price of the 8080 had dropped from its original $400 – $500 level to one that permitted a number of small companies (many working out of basements and garages) to begin shipping reasonably priced microcomputers, mostly as kits, to a growing audience of enthusiastic hobbyists. Although you couldn't do much more than programming for grins in 8K of memory, it became obvious very quickly that one important element was missing: an operating system capable of supporting disk file management.

The decision of several companies not to develop their own operating system and the inability of others to deliver a suitable product were primarily responsible for CP/M's becoming today's pseudo-standard. Several of the first companies to adapt CP/M for their products made disk drive subsystems for S-100 bus-compatible machines. The prominence of the S-100 bus at this time focused even further emphasis on CP/M and detracted from other specialized research and development projects.

The average early microcomputer sported a 2 MHz clock rate and generally used less than the maximum 64K of main memory, placing obvious limitations on the resources available to CP/M. This limited environment helped sharpen the basic design goal of CP/M: to provide a straightforward, no nonsense approach to a single-user operating system.

Within just a few years, CP/M was offered on practically every 8080-compatible 8-bit computer system built and was being adapted or at least planned for every new system to come along. This nearly universal acceptance of CP/M heightened the need for enhancements to the disk interface portion of the BDOS to ease the adaptation process.

Version 2.2 of CP/M was brought out in the early '80s. Its most exciting feature was a new, flexible, table-driven generic disk interface and the ability to address larger capacity disk drives. Not only was the job of integration greatly simplified, but the flexibility of the new interface allowed many new storage options to be offered by the computer manufacturers. It soon became common to offer several different disk capacity options for the same computer.

Within the last year, DRI has released its latest revision of CP/M called CP/M Plus. This version turns out to be a completely new system, structured for an environment that includes expanded memory of at least 96K. As expected, when given enough memory to perform all of its magic, CP/M Plus can improve the runtime of programs that make heavy use of the disk system.

For over 10 years, CP/M has led the way in 8-bit operating systems; it continues today to maintain its position of prominence by having more installed systems than any competitor. Since those early days, CP/M has gone through a number of upgrades but no change in philosophy. It remains the same rock-solid generic disk operating system that we have grown up with.

How It Operates

CP/M is simple-minded in its dealings with the host I/O system (BIOS). The rules are few; as long as they are religiously followed, the marriage will remain a peaceful one.

The portion of CP/M that intercedes between the computer's hardware and the BDOS is the BIOS. Contained in the BIOS are all the machine-dependent routines needed to interpret CP/M's language to that of the hardware system. One major task of the BIOS, and probably the most important, is to maintain peace and order over the disk system.

Most of the time during a disk I/O operation is spent waiting for the drive's mechanical apparatus to properly position itself for the data transfer. Consider for a moment the number of interrelated mechanical events that must happen in precise order to prepare for a single data transfer. First, the spindle motor is started and allowed to stabilize at a constant speed.

Next, the heads are loaded against the disk surface. Finally, the heads are stepped in or out to the proper track. At this point, to search for the requested sector and transfer it requires only a few additional milliseconds.

To complete the startup cycle and read one record on even the latest model disk drive can account for a delay of as much as one-third to one-half of a second. If this seems like a long time, it is—certainly long enough to be worthy of efforts to reduce it to a bare minimum. It's not hard to estimate how slowly a program would run if a complete startup cycle were necessary for each data record read. It stands to reason that as much data as possible should be transferred during each cycle.

The complexity of the BIOS is largely dependent on the hardware and whether the physical disk records are larger than CP/M's logical sector size of 128 bytes. If, for example, the disk being used is formatted with a physical sector size of 128 bytes, the BIOS has the fairly uncomplicated job of instructing the disk controller hardware where to put each data sector as directed by the BDOS. Depending on the hardware, this task can be as simple as loading several registers with the track and sector values and initiating the I/O operation by loading an instruction into the command register. If you are using one of the newer intelligent disk controllers that address the disk through data block numbers rather than by actual sector and track numbers, the job is slightly more complicated; some calculations are necessary but only a few.

The introduction of CP/M 2.x brought with it the ability to easily use physical sector sizes larger than the standard logical sector size of 128 bytes. At this time, two new buzz words became prominent when referring to disk systems: single-density describes disk formats in which both the logical and the physical sector sizes are 128 bytes in length, and double-density is used to describe almost any disk format where more than one logical sector is contained in a physical sector. This new version of CP/M also made it easier for the system integrator to fine tune the disk system to the computer system.

There are two reasons for complicating what was once the very simple issue of disk I/O. Changing the physical disk sector size from 128 to 512 bytes, for example, should in theory increase the disk system's throughput by a factor of four because four times as much data is being moved in one operation: for each sector read from the disk, the next three reads should be satisfied directly from memory and at memory speed. In theory this may be true, but other factors generally prevent an increase of this magnitude. A goal of twice as fast is probably more in order.

Another benefit of using larger disk sector sizes is that you can store more data on the same size disk. Most of the recording area of a disk is used for storage of control information that separates the physical sectors and describes the contents of the data record. For example, each data record area is preceded by a series of sync bytes to assist the hardware in locking onto the recorded signal. Once in phase with the recorded signal, the data record descriptors, the data record, and a few error-checking bytes are read from the disk. To increase the data content in each sector from 128 to 512, for example, would require reducing the number of sectors and slightly changing the amount of control information and its content.

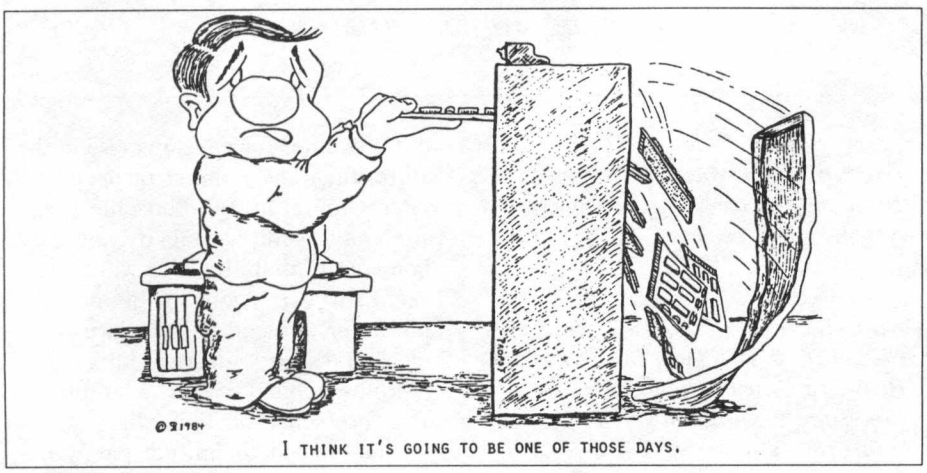
A perfect example of this is the double-density disk format used by Intercontinental Micro Systems (ICM). The standards published by Western Digital (ICM uses its 2793 disk controller chip) specify that, to ensure reliable operation, a maximum of 15 – 512 byte sectors be allocated to each track of an 8-inch disk. After exhaustive testing, ICM found that it could reliably format each track with 16 – 512 byte physical sectors. Formatting its double-density disks in this way permitted ICM to achieve the maximum possible disk capacity without using the extra memory required by an even larger sector size.

Using Large Sectors

To use physical sectors larger than 128 bytes requires that two routines be present in the BIOS. The first routine, blocking, maps CP/M's logical sector requests into the larger host buffer. The other routine, deblocking, performs the opposite operation of extracting the proper logical sector from the host buffer.

Imagine for a moment that a memory area 512 bytes in length has been set aside and divided into four 128-byte increments. Each of the 128-byte increments corresponds to a CP/M logical sector, and the entire 512-byte area corresponds to the host physical sector. When CP/M makes its first logical write request, the data is moved from the DMA address to the first slot in the host buffer. The second CP/M write request is placed into the second position of the host buffer, and so on until the fourth and last logical sector slot has been filled; the host buffer now must be emptied or data will be destroyed. From this example, we can see that using larger physical sectors allow the number of actual disk I/O requests to be reduced.

When CP/M makes a logical sector read or write request, the request is accompanied by the actual sector and track numbers. These two values are all that is needed to calculate exactly



I THINK IT'S GOING TO BE ONE OF THOSE DAYS.

(Continued on page 94)

Varieties of Unix

An Introductory Guide to Microcomputer Unixes

Now available on computers from dozens of manufacturers ranging from Altos and Apple to Zentec and Zilog, including systems as small as the IBM PC and as powerful as the Cray supercomputer, Unix is well positioned to become a widely accepted standard operating system. According to a recent report issued by *Montgomery Securities*, "It is difficult to underestimate the importance of Unix to the computer industry . . . . In the business microcomputer world, Unix will simply become the industry's standard operating system. MSDOS will evolve to be Unix compatible and will be available as a subsystem under Unix."[1]

By the time you acquire a general understanding of why Unix microcomputers are so attractive, you are likely to have discovered that there are several varieties of Unix, plus several "Unix-like" or "Unix look-alike" systems. What are the differences between these systems, and do the differences make any difference?

There is no standard Unix. It's not even clear exactly what programs are defining features in any given implementation. Is it possible to extract a hypothetical standard from the union or intersection of existing implementations?

True versus Imitation Unixes

To qualify as true Unix, an operating system must be licensed by AT&T, owner of the Unix trademark. True Unix systems are developed by purchasing a tape containing Unix source code from AT&T and enhancing and massaging the AT&T programs in accordance with the requirements of the intended environment. For example, a user-friendly front end menu system might be added to shield novices from the possibly intimidating terseness of the standard Bourne shell user interface.

Imitation Unix systems, such as Coherent, Cromix, Idris, QNX, uNETix, and UNOS, mimic Unix but do not contain the AT&T source code. Given the momentum now enjoyed by true Unix, the advantages, if any, of an imitation Unix ordinarily cannot compensate for the dangers involved in traveling along a nonstandard path. Of course, it all depends on your requirements: if you are doing nothing but word processing, and the only Unix feature you care about is the Unix tree-like hierarchical file structure, and are sure you won't want to use your computer for anything else in the future, then any number of operating systems imitating the Unix file structure could suffice. But if you care about the availability of a wide selection of software, want to stay flexible, and prefer to avoid unnecessary risks, you should stick with true Unix.

Cromemco's Cromix exemplifies some problems of look-alikes. Cromemco decided to imitate partly be-

Alan Walworth, Fortune Systems Corporation, 101 Twin Dolphin Drive, Redwood City, CA 94065.
Copyright © Alan Walworth 1984. All rights reserved.

cause Unix licenses were very expensive at the time (the price has since plummeted) and partly because major modifications, such as translating everything into Z80 assembly language and eliminatin- many capabilities, were needed to accommodate the limitations of an 8-bit 64K Z80 environment. Cromemco did a good job of fitting many Unix features into that environment: back in 1981, when it was first released, Cromix was impressive. In 1984, however, powerful, new, low-cost hardware makes it unnecessary to accept the constraints of an 8-bit processor with a 64K address space.

By using Z80 assembly language rather than the high level C language, Cromemco lost a major advantage of Unix: efficient portability to new hardware. Thus moving Cromix to the 68000 was a long and difficult task.

Except for software using Unix features that Cromix failed to implement, porting Unix software to the 68000 Cromix is feasible. The task of developing a Cromix version, however, typically ranks low (if it appears at all) on a Unix software developer's list of priorities; the result is that a person who opts for Cromix can make use of only a small fraction of the Unix software available to the owner of a true Unix system.[2] (In July 1984, Cromemco acknowledged the importance of standard Unix by unveiling new computers that run Unix System V.)

Because the Unix imitations, on the face of it, are not in the running, and because an analysis sufficiently detailed to provide insight into which ones might be worth considering in which special circumstances would have to be quite lengthy, the look-alikes will not receive further attention here.

A brief history of Unix will be presented to set the stage for a discussion of current microcomputer Unixes. Stanix, a hypothetical mainstream bare-bones version of Unix, then will be described in some detail, not only to provide general information about Unix but also to illustrate what is involved in analyzing a Unix system and to provide a paradigm useful as a starting point for describing various Unixes. Fortune Systems' FOR:PRO, Xenix, PC/IX, VENIX, System V, and 4.2 BSD will be examined next. Although this survey includes most of the main microcomputer Unixes, some significant versions are regrettably omitted due to lack of sufficient information about them at the time of writing. Vendors annoyed by inadequate coverage here are invited to provide details about their Unix implementations to the author, so that a more comprehensive survey can be provided in a later article.[3]

A Brief History of Unix

Unix was developed in the early 1970s at Bell Labs by Ken Thompson and his associates. From the outset, the intent was to create a convenient and flexible environment for program development. Contrary to the tendency at that time to focus on maximizing the efficiency with which expensive hardware could be utilized, the developers of Unix concentrated their attention on efficient use of human resources (and, in particular, on efficient use of software developers' time). As hardware costs plummeted and software development costs rose, the wisdom of that approach became increasingly apparent.

Prior to Unix, operating systems had been written in assembly language, and Unix itself was originally written in assembly language for the DEC PDP-7 processor. In 1972, however, Unix was largely rewritten in C, a new language developed at Bell Labs by Dennis Ritchie, that combines low-level power with high-level convenience and portability. (A small part of the Unix code—about 5 percent—remains in assembly language for the sake of efficiency and because of the occasional need to use a hardware function not accessible via C.) Although programming the operating system in a high-level language makes its use of the hardware somewhat less efficient, that loss in efficiency is more than offset by the relative ease with which the code can be understood, maintained, enhanced, and ported to new hardware.

The distribution of Unix was constrained by federal restrictions on the AT&T monopoly's participation in the commercial marketplace; in 1973, however, Unix Version 5 (not to be confused with System V) was released to educational institutions and to some commercial organizations. PDP-11 minicomputers were widely used in universities at that time, and Unix rapidly became popular as an operating system for those machines.

Version 6, released in 1975, was made available to commercial establishments as well as nonprofit organizations, but the price for the commercial market was very high, documentation was minimal, and support and maintenance were not provided.

Version 7, which appeared in 1978, contained enhancements including support of large files (up to one billion bytes), a standard I/O library, a more capable C compiler, an improved shell, and more sophisticated typesetting software. The University of California at Berkeley, Brian Kernighan's alma mater,[4] ported Version 7 to its VAX minicomputers, and that was the configuration that was soon widely used at universities and other noncommercial institutions.

Bill Joy and his associates at the Berkeley Computer Science Department introduced a wide variety of enhancements, including the vi screen editor, the C shell, curses, and termcap. Berkeley enhancements were made available to the outside world in the 4.1 Berkeley Software Distribution (4.1 BSD). A more recent Berkeley version of Unix, known as 4.2 BSD, offers virtual memory, networking capabilities and faster file access.

System III, AT&T's first serious attempt to market Unix as a product, appeared in 1981. System III contained features from the Programmer's Workbench (PWB), which includes utilities such as the Source Code Control System (SCCS), Remote Job Entry (RJE), and nroff and troff.

System V, announced in 1983, is the version of Unix most widely used within the companies that once constituted AT&T and it is this version that the reconstituted AT&T would like to see accepted as a universal standard. System V contains many of the 4.1 BSD enhancements. It features Interprocess Communication (IPC), which employs named pipes, messages, shared memory, and semaphores. In January of 1984, AT&T announced a new release of System V, V.2.

Constant versus Variable Features

One of the reasons for Unix's populari-

ty is that it contains features now widely recognized to be essential for efficient operation of a multi-user system. The hierarchical file structure allows sensible organization of data. The file permission system protects files from unauthorized access or destruction. The password system limits use of the system to legitimate users. Convenient background execution improves user productivity by making it easy to complete time-consuming data processing tasks without interfering with ongoing work.

These basic features are explained in numerous introductory Unix texts, are fairly well known, and do not differ significantly from one version of Unix to another. Therefore, although they are extremely important, they will not be discussed further here. (For an explanation of such Unix fundamentals, see *Understanding Unix* by Groff and Weinberg, *The Unix Operating System* by Christian, or *The Unix Operating System* by Bourne.)

The following sections concentrate on two aspects of Unix that vary far more from version to version and are much more difficult to grasp: utilities and system calls. It is difficult to become familiar with these areas of Unix due to the vastness of the territory they cover. A typical Unix system has hundreds of utilities and dozens of system calls. Mastering many utilities requires a substantial effort because of their complexity, and understanding what certain system calls do (or even what a system call is) requires an understanding of internal computer operations more advanced than that enjoyed by many users.

The Utilities of Stanix

We will examine the utilities first because they are of primary interest to the majority of Unix users. The limited space available here makes it impossible to explain the utilities thoroughly, so we will simply list the most prevalent ones with a brief indication of what they do. The resulting checklist can serve as a starting point for detailed analysis of the utilities provided by various versions of Unix. Our immediate objective, however, is to use this list as a means of revealing, in general terms, both the contents of the Unix toolkit and the extent to which

acctom	searches and prints process accounting files
adb*	general purpose interactive debugger
admin	creates and administers Source Code Control System files (SCCS is a set of utilities for the administration of software development or document development projects. SCCS enables you to keep track of changes made to source code or text files, including the date of each change, the nature of the change, who made the change, etc.)
ar*	maintains archives and libraries
arcv	converts archives to a new format
as	assembler (all versions have assemblers, but they're not all called *as*)
at	executes a command at a specified future time
awk	a language for pattern scanning and processing
banner	prints large letters
basename*	removes "/" and "." extensions from a filename or path name
bc*	interactively processes arbitrary-precision mathematical expressions
bdiff	reports differences between two big files
bfs	scans big files
cal*	prints a calendar
calendar	reminder system
cat*	catenates (i.e., prints or lists) one or more files
cb*	C program beautifier
cc*	C compiler
cd*	changes the current directory
cdc	changes the delta commentary of a SCCS file
chgrp	assigns a file to a different user group
chmod*	changes file access permissions
chown*	changes the ownership of a file
chroot	changes the root directory for a process
clri	clears an inode
cmp*	compares two files
col*	takes reverse line feeds out of a file
comb	combines SCCS deltas
comm*	reports lines common and uncommon to two sorted files
cp*	copies a file or set of files
cpio	copies file archives in and out
cref	generates C program cross-reference listings
cron	executes commands contained in /etc/crontab at predesignated times
crypt*	encrypts or decrypts a file
csh	Berkeley's C shell
csplit	context file split
ctags	creates a function name index for a C or Fortran 77 source file
cu*	interactive system for calling another computer and transferring text files or for functioning as a terminal on the called system
cut	cuts out selected fields from each line of a file
cw	prepares constant width text for troff
date*	displays or sets the system date and time
dc*	interactively processes arbitrary-precision mathematical expressions
dcheck	checks the consistency of file system directories
dd*	converts and copies a file (e.g., to a non-Unix system)
delta	changes a SCCS file
deroff*	removes nroff, troff, tbl, and eqn constructs

Table I.
Stanix Utilities

df	displays statistics on disk usage
diff*	shows how two files differ
diff3*	reports differences among three files
diffmk	marks differences between files
dircmp	compares directories
du*	reports disk usage and file size statistics
dump	dumps selected parts of an object file
dumpdir	prints the names of files on a dump tape
echo*	displays arguments such as strings, filenames, shell variables, and command output
ed*	the old standard Unix line editor, useful in shell scripts
efl	extended Fortran language
egrep*	searches files for matches to a full regular expression
env	sets the environment for command execution
eqn	formats mathematical text for nroff or troff
ex	Berkeley's improved version of the ed editor
expr*	evaluates simple mathematical expressions and extracts strings
f77	Fortran 77 compiler
factor	factors a number
false*	returns a false value
fgrep*	searches files for matches to a fixed string
file*	reports the type of a file (executable code, text, etc.)
find*	locates files with specified properties
fsck	file system check with automatic repair option
get	gets a version of a SCCS file
getopt	parses command options
gets	suspends shell script processing to get user input
getty	sets the terminal mode and baud rate during startup
graph	draws a graph
grep*	searches for a text pattern in one or more files
head	displays the first lines of a file
help	provides helpful information about the SCCS
hyphen	finds hyphenated words
icheck	checks file system consistency
id	prints user and group IDs and names
init*	sets the environment for all user programs and allows users to log on to the system
join*	produces a join of two relational data base files
kill*	terminates designated processes
ld*	link editor
lex*	generates lexical analyzers
line	reads a line from standard input
lint*	reports possible problems in C programs
ln	makes a link to a file
login	logs the current user out and logs in a new user
logname	gets the current login name
look	finds lines in a sorted list
lorder	finds the ordering relation for an object library
lpq*	displays the printer queue
lpr*	spools a file for printing on a specified printer
lprm*	removes an item from the printer queue
ls*	displays a list of files in a directory
m4	a macro processor
mail*	sends mail to other specified users
make*	maintains sets of related program files
makekey	creates an encryption key
man	displays Unix Manual pages

Table I (cont)

various versions of Unix are the same at the utility level.

Utilities are simply programs that serve as useful tools. According to the Unix philosophy, each utility should perform a single task well. Power and productivity result from creatively combining the basic functions provided by the utilities. Some tasks performed by Unix utilities, such as copying or removing a file or logging in and out, are fundamental necessities for operating the computer. Thus, they are naturally regarded as part of the operating system (by definition an operating system is software that performs such fundamental functions). Traditionally, however, many other programs that are far less essential to operation of the computer have been regarded as part of the various Unix operating systems. To some extent, this is just a result of all programs that AT&T decides to provide on the Unix release tapes becoming automatically part of Unix. There is no other reason why Fortran, for example, is considered part of Unix and COBOL is not.

The Stanix utilities described in Table I (page 26) are standard in the sense that each appears in at least three of the following six major versions of Unix: Version 7, 4.1 BSD, System III, System V.2, FOR:PRO, and Xenix 3.0. This selection procedure, although somewhat arbitrary, produces a list of the mainstream utilities while winnowing out exotic commands like rp6fmt, vpmc.u3b, 300s, and 4014, which are of no importance to most Unix users. Of the 195 commands selected by this procedure, 92 are found in all six of the Unix versions; these are indicated in the table with asterisks.

Stanix System Calls

System calls are to a program what utilities are to the user: they are commands used by the program to tell the operating system to perform some basic action, such as opening a file for reading or creating a new process. Programs can also interact with the operating system by calling functions. The difference between the two methods is that the code for system calls is embedded in the operating system itself, whereas the code for a function must be obtained from outside the operating system.

As in the case of utilities, the list of

Stanix system calls in Table II (page 30) was derived by selecting those found in at least three of the following Unix systems: Version 7, 4.1 BSD, System III, *System* V.2, FOR:PRO, and Xenix 3.0. As before, asterisks indicate inclusion in all six of these Unixes.

Comparison of Six Major Versions of Unix to Stanix

Having presented Stanix, we can now examine the extent to which various versions of Unix differ from this standard. For each of the six major Unix versions—Version 7, 4.1 BSD, System III, System V.2, FOR:PRO, and Xenix—Table III (page 32) shows the number of Stanix and non-Stanix utilities contained in each version and the number of Stanix and non-Stanix system calls in each version.

The figures in Table III should not be used as a means of evaluating these versions of Unix. These figures are presented only as a method of conveying a general idea of the extent of these systems, the degree to which they overlap, and how much they differ. Although much effort has been devoted to arriving at accurate figures, inaccuracies are inevitable.

A major problem in compiling such statistics is that available listings of commands and system calls either contain omissions or, as in the case of complete sets of Unix manual pages, are too voluminous to digest in the time that was available for this study. The *Man Pages* tables of contents (the chief source used here for Version 7, 4.1 BSD, System III, and System V) fail to reference commands like egrep and fgrep, which are documented on the same man page as grep. Obvious omissions of this sort have been corrected, but others have probably slipped through.[5] In particular, it is likely that the number of additional system calls shown in Table III for these systems is misleadingly low.

The number of additional utilities included is less meaningful than it might seem at first glance; some additional available commands were not counted because they are not considered *operating system* commands. This is especially true of FOR:PRO and Xenix. For example, communication utility programs available from AT&T for use on System V tend to become

mesg*	controls whether other users can write to your screen
mkdir*	makes a directory
mkfs*	constructs a file system on an unmounted device
mknod*	makes a device or special file
mm	prints documents using the mm macros
mmt	typesets documents, slides, and viewgraphs
more	displays a file one screen at a time
mount*	mounts a file system
mv	moves a file or set of files
ncheck*	displays a table of path names and inodes for a file system
newgrp*	changes your group identification
nice*	sets the priority at which a program should run
nl	line numbering filter
nm*	displays object file symbol names
nohup	executes a command that will ignore keyboard interrupts and will continue after the user logs out
nroff	a text formatter
od*	displays a file in requested formats such as octal, hex, and ASCII
pack	compresses and uncompresses files
passwd*	assigns or changes an account's password
paste	merges the same lines of several files or successive lines of a single file
pr*	paginates and adds an optional header
prof*	profiles program execution
prs	prints the deltas of a SCCS file
ps*	displays information about current processes
pstat	displays a table of system status information based on the state of the kernel
ptx	permuted index generator
pwd*	prints the working (current) directory name
quot	summarizes the file system ownership
ranlib	converts an archive file to a random library format that can be loaded efficiently
ratfor*	rational Fortran preprocessor
rc	invoked by init to run during startup and shutdown
regcmp	regular expression compiler
restor	restores the file system incrementally
rm*	removes a file or set of files
rmdel	removes a delta from a SCCS file
rmdir	removes a directory
sact	displays current SCCS editing activity
sccsdiff	compares two versions of a SCCS file
sdb	symbolic debugger
sdiff	side-by-side difference program
sed*	stream editor
setmnt	establishes a mnttab table
sh*	the standard Bourne shell
shutdown	the normal system shutdown program
size*	reports object file size statistics
sleep*	suspends shell execution for a specified interval
sort*	sorts lines of a file by specified fields
spell	checks spelling
spline	interpolates a smooth curve
split*	splits a file into pieces
strings	locates printable strings in a binary file
strip*	removes symbol table and relocation bits

Table I (cont)

struct	Fortran to RATFOR translator
stty*	displays and sets terminal characteristics
su*	temporarily changes your user ID so you can access restricted files and programs (The password must be given or the command will fail.)
sum*	calculates a checksum for a file to detect bad blocks
sync*	updates the super block on the hard disk and ensures that all disk writes have been completed
tabs	sets terminal tabs
tail*	displays the last lines of a file
tar*	tape archive utility
tbl	formats tables for nroff and troff
tc	typesetter simulator
tee*	copies standard input data, transferred through a pipe, to a specified file and to the standard output (A pipe passes the output of one command to the input of another.)
test*	tests to see if a condition exists or if a relational expression is true or false
time*	displays time statistics for the execution of a command
touch*	accesses a file without changing it to update its "date of latest access" field
tp	manipulates a tape archive
tr*	translates or filters specified characters
troff	formats text for typesetting
true*	returns a true value
tset	sets shell variables to accommodate a specific type of terminal
tsort*	sorts contents of a file based on a partial ordering of items in the file
tty*	displays the name of the current terminal's device file
umask	sets the permission default mask used in file creation
umount	unmounts a file system
uname	prints the name of the Unix system being used
unget	reverses a get on a SCCS file
uniq*	eliminates successive duplicate lines found in a file
units	measure conversion program
update	periodically updates the super block on the hard disk with sync to minimize data loss in the event of a system crash
uuclean	cleans up the uucp spool directory
uucp*	allows sophisticated automated communication between Unix systems
uux*	executes a command on a remote Unix system
val	determines if a file is a SCCS file with specified characteristics
vc	converts lines of input using specified arguments and control statements
vi	a sophisticated screen editor developed at Berkeley
vpr	Versatec printer spooler
wait*	stops interactive shell processing until all background processes are completed
wall*	broadcasts a message to all logged on users
wc*	reports the number of characters, words, lines, and pages in a text file
what	identifies SCCS files
who*	shows who is logged on
whodo	reveals who is doing what on the system
write*	allows interactive communication with other logged on users
xargs	constructs argument lists and executes commands
yacc*	yet another compiler compiler: a compiler creation system
yes	repeatedly displays a string or argument

Table I (cont)

part of the operating system by fiat, but when such a utility (e.g., a VT100 terminal emulator) is offered by Fortune, it is not considered part of FOR:PRO.

Finally, bear in mind that in some cases two versions of Unix have a utility (or system call) with the same name, but the functions performed, or the options available governing just what can be done with the utility or system call, are different.

With all their shortcomings, the numbers are still of interest. They show that the major Unix versions share a large body of commands that are essentially the same from one version to another. The body of shared utilities is sufficiently large that people familiar with one version will feel at home with another. They, however, may be annoyed at the lack of favorite utilities: programmers accustomed to Berkeley versions, for example, are likely to be disturbed when they find the C Shell (csh) is missing from the AT&T versions.

This analysis of the extent to which major Unixes overlap suggests two guidelines for the evaluation of any particular version of Unix: (1) if the version does not contain a healthy majority of the 195 Stanix utilities, it is, on the face of it, seriously incomplete, and (2) it is important to check to be sure that all the utilities you will need are available. Of course, novices who have no idea what utilities will be needed and who intend to avoid direct interaction with the operating system can apply the second guideline only by relying on expert assistance, the recommendations of experienced users, or the general reputation of the product to establish that the system will be able to perform as required.

Having gotten a feel for the extent to which these main versions of Unix differ from one another, let's turn to a consideration of the noteworthy features of various microcomputer Unixes. FOR:PRO will be discussed at some length to illustrate the sort of customization of Unix that is needed to create an effective and friendly microcomputer operating environment. We'll then look at other important microcomputer Unixes, including some that could not be included in the analysis above due to lack of data.

FOR:PRO

Fortune's FOR:PRO is Version 7 Unix with some System III software, 4.1 BSD enhancements, and Fortune enhancements. Thus, FOR:PRO is a variant of mainstream Berkeley Unix. Like System V, it contains many of the most desirable Berkeley enhancements.

In addition to Berkeley enhancements, FOR:PRO contains enhancements made by Fortune. These are of three kinds: bug fixes, changes for improved performance, and alterations needed to adapt Unix to a microcomputer environment. We will examine briefly the major changes of the last kind, beginning with alterations related to the Fortune computer's floppy disk drive.

The Unix source code provided on AT&T's distribution tapes is designed for use on a minicomputer that employs tape drives for backup and initial loading of programs. Although a tape backup unit is available for the Fortune micro, the floppy disk drive included with the base system has to provide backup capability for users who don't have the tape streamer; the floppy drive is also normally used for new software installation. Thus, FOR:PRO needs driver software that can interface with the floppy drive.

More interestingly, it needs a version of the cp (copy) command that, unlike the standard Unix cp, is intelligent enough to pause and suitably prompt the user when a new floppy disk needs to be inserted during backup of a set of files too large to fit on a single disk. Similarly, when restoring a file system from a set of floppies, the Fortune cp needs to know enough to prompt for insertion of the next disk. A less essential Fortune enhancement to cp is the recursive option, which simplifies backup onto floppy disks by enabling a single command to be used to copy an entire file system, including the contents of subdirectories.

Unix systems normally require reconfiguration when hardware components such as new memory boards or I/O processors are added. In other words, the operating system must be told about the new hardware and reconstituted to take the new capabilities into account. Some Unix micros lack reconfiguration capabilities, making hardware additions impossible.[6] On others, reconfiguration procedures require expertise that new users lack. Fortune avoids this difficulty with auto-configuration: when the Fortune system is powered up, it automatically determines what hardware is in the system and configures itself appropriately. For users with sufficient expertise, additional facilities for fine-tuning the system will optimize performance based on the nature of expected system usage.

A third major Fortune enhancement is a menu shell that enables novices to use applications programs and administer the system without having any knowledge of Unix. Although the menu system is helpful for beginners, experienced users find it easier to work directly with Unix using either the Bourne shell or the C shell. As users learn more about Unix, they can easily switch back and forth between the Bourne and menu shells, letting the menu system assist with tasks they do not yet know how to perform with regular Unix commands.

As a rule, business microcomputer users are far less tolerant of operating system problems than engineers and computer scientists. For the latter sort of user, a system crash may be a minor annoyance; for the business user, it is more likely to be a major trauma. This is the case both because downtime and lost data can be very costly and because the business user typically has no idea what to do when something goes wrong.

Two years of selling Unix systems to the business marketplace has given Fortune incentive to get the glitches out of its version of Unix. The result is that FOR:PRO is now an exceptionally robust Unix implementation. For the same reason, FOR:PRO now comes with a set of documentation oriented to the needs of inexperienced users.

access*	reports the accessibility of a file based on its permission modes and the real user ID of the user
acct*	initiates records for each system process in a file, or disables the record-keeping mechanism
alarm*	causes a signal to be sent to the current process after a specified number of seconds
brk	changes the amount of memory accessible by the current process
chdir*	changes the current working directory
chmod*	changes the access permission modes of a file
chown*	changes the owner and group of a file
chroot	changes the relative root directory for file identification
close*	closes a file
creat*	creates a new file
dup*	returns another file descriptor for a previously opened file so that the file can be accessed by two different file descriptors
exec*	executes an executable file
exit*	terminates a process
fcntl	provides control over open files
fork*	creates an identical twin (child) process
fstat	returns file status
getpid*	returns the process ID of the current process
getuid*	returns the real user ID of the current process
ioctl*	controls the operation of a terminal or other character device
kill*	sends a signal to a specified process, often resulting in termination of that process
link*	assigns an additional name to a previously existing file
lock	hastens execution by virtually preventing the current process from being swapped out of memory
lseek*	changes the position of the read/write pointer in a file
mknod*	creates a directory or device file
mount*	mounts or unmounts a file system

Table II.
Stanix System Calls

Xenix

Xenix is Microsoft's adaptation of Unix System III. Like FOR:PRO, Xenix includes ex and vi; it does not include an office oriented word processor comparable to Fortune's Fortune:Word.[7] For those wishing to adhere to the traditional Unix approach to word processing—an approach more suited to program documentation and writing scientific articles than to office automation—Xenix provides tbl for creating tables and eqn for representation of complicated mathematical expressions.

Also offered are the diction and style programs, which search for awkward expressions, provide a measure of how difficult your writing is to read, and so on. Xenix provides a novice-friendly menu system called the "Visual Shell." Because it's also available for MSDOS systems, this shell offers a means of providing a consistent user interface on all micros in an environment containing both Unix systems and IBM PCs.

In addition to uucp, Xenix includes Microsoft's micnet communications software, a less elaborate system that is somewhat easier to use than uucp. Unlike FOR:PRO, Xenix does not provide on-line documentation. However, a help facility is included in the visual shell. Xenix 3.0 provides utilities for reading and writing MSDOS files, a convenient feature for anyone who wants to run both MSDOS and Xenix on the same computer.

PC/IX

PC/IX is a fairly complete version of Unix System III, developed for the IBM PC by Interactive Systems Corp. of Santa Monica. PC/IX is said to be fast and to contain a good reconfiguration capability, but it is available only as a single user system and lacks some standard Unix utilities. The missing programs include fsck, pstat, ranlib, tar, and the C shell, csh. Although lpr is not included, a sophisticated print spooling utility called print is supplied instead. INED, a good screen editor derived from the Rand e editor, is offered in place of vi.

Utilities are provided for moving files back and forth between PCDOS and PC/IX. No menu shell is offered. The troff supplied with PC/IX produces output for only a single phototypesetter, the Graphics Systems CAT. Compilers are available for C and Fortran 77; interpreters are included for BASIC and a version of SNOBOL.

VENIX

VenturCom's VENIX is an IBM PC implementation of Version 7 Unix with Berkeley enhancements, including vi and the C shell. Unlike PC/IX, VENIX can handle up to three simultaneous users. Awk, lex, and yacc are included, but lex is apparently incomplete.[8] Troff is absent. Simple graphics functions are provided for tasks such as drawing lines and circles. On-line reference materials, available on full-blown Unix systems via the man command, are not included in VENIX. A BASIC interpreter is provided. Semaphores are available for interprocess communication, a step in the direction of System V's IPC capability. The Final Word is supplied as an optional word processing alternative. Of special interest to people dreaming of portable Unix systems is the plan (perhaps a reality by the time this is published) to offer a version of VENIX as an option on Data General's new 10-pound PC.

System V

In addition to bug fixes and performance enhancements, System V contains a variety of changes from System III. Improved file I/O performance may be noticeable due to a doubling of the block size to 1024 bytes.[9] An improved scheme of file system updates reduces the risk of file corruption in the event of a system crash.

From a software developer's viewpoint, the most interesting improvement is System V's new Interprocess Communication (IPC) capability, which utilizes messages, shared memory, semaphores, and named pipes. The message capability allows a process to directly and efficiently communicate with another process. Shared memory

nice*	sets the execution priority of the current process
open*	opens a file for reading, writing, or appending
pause*	suspends execution of the current process until a signal is received
pipe*	allows two processes to communicate by creating a mutually accessible data buffer
profil*	monitors or disables the user's program counter during the execution of a C program
ptrace*	traces and controls a child process
read*	reads data from a file
setpgrp	sets the process group ID
setuid*	sets the effective user or group ID of the current process
signal*	intercepts signals for analysis, often preventing termination of the current process
stat*	reports the attributes of a file
stime*	sets the system date and time
sync*	writes all important memory data to disk
time*	returns total seconds since January 1, 1970
times*	returns system time consumption statistics for the current and child processes
ulimit	gets and sets user limits
umask*	sets the default mask used to establish access permissions for files
umount	unmounts a file system
uname	returns the name of the current Unix system
unlink	unlinks a filename from a file, and if it is the last filename or link to the file, removes the file from the file system
ustat	returns file system statistics
utime*	changes the record of the latest time at which a file was accessed and modified
wait*	suspends execution of the current process until a child process terminates
write*	writes data into a file

Table II (cont)

is memory available to more than one process. Semaphores are used to coordinate access to shared memory by the multiple processes using it.[10] Named pipes, like ordinary pipes, channel data from one process to another. The difference is that an ordinary pipe requires the two processes to be concurrent; with a named pipe, output from a current process can serve as input to a process that does not yet exist. An IPC remove command is available to clear out unwanted message queues and semaphore identifiers. A new file system checker, dfsck, allows multiple file systems to be checked simultaneously.

The C programming environment has been altered in several respects, including a new Common Object File Format (COFF), enhancements to the math library, and cflow, a new program for flow analysis that generates module-calling hierarchy charts. An annoying deficiency of the System V C compiler is that it allows variable names to be only eight characters long (a limitation shared by many older C compilers). The improved symbolic debugger, sdb, can be used for both C and Fortran 77 programs.

The –ms macros, used by the nroff text formatter, have been eliminated.

4.2 BSD

4.2 BSD, the latest release of Unix from Berkeley, offers virtual memory, advanced networking capabilities based on "sockets," enhancements for fast file access, and the ability to use filenames much longer than the 14-character maximum on most Unixes. 4.2 BSD is the Unix of choice for engineers using relatively large and powerful systems such as superminis; it is less suitable for ordinary micros. Systems built around the 68000 processor, for example, lack the hardware required for effective virtual memory implementation (the 68010 overcomes this lack). Also, 4.2 BSD is relatively large, which interferes with efficient performance on a micro with limited memory. As micros become more powerful, these problems will become less significant.

The Future of Unix

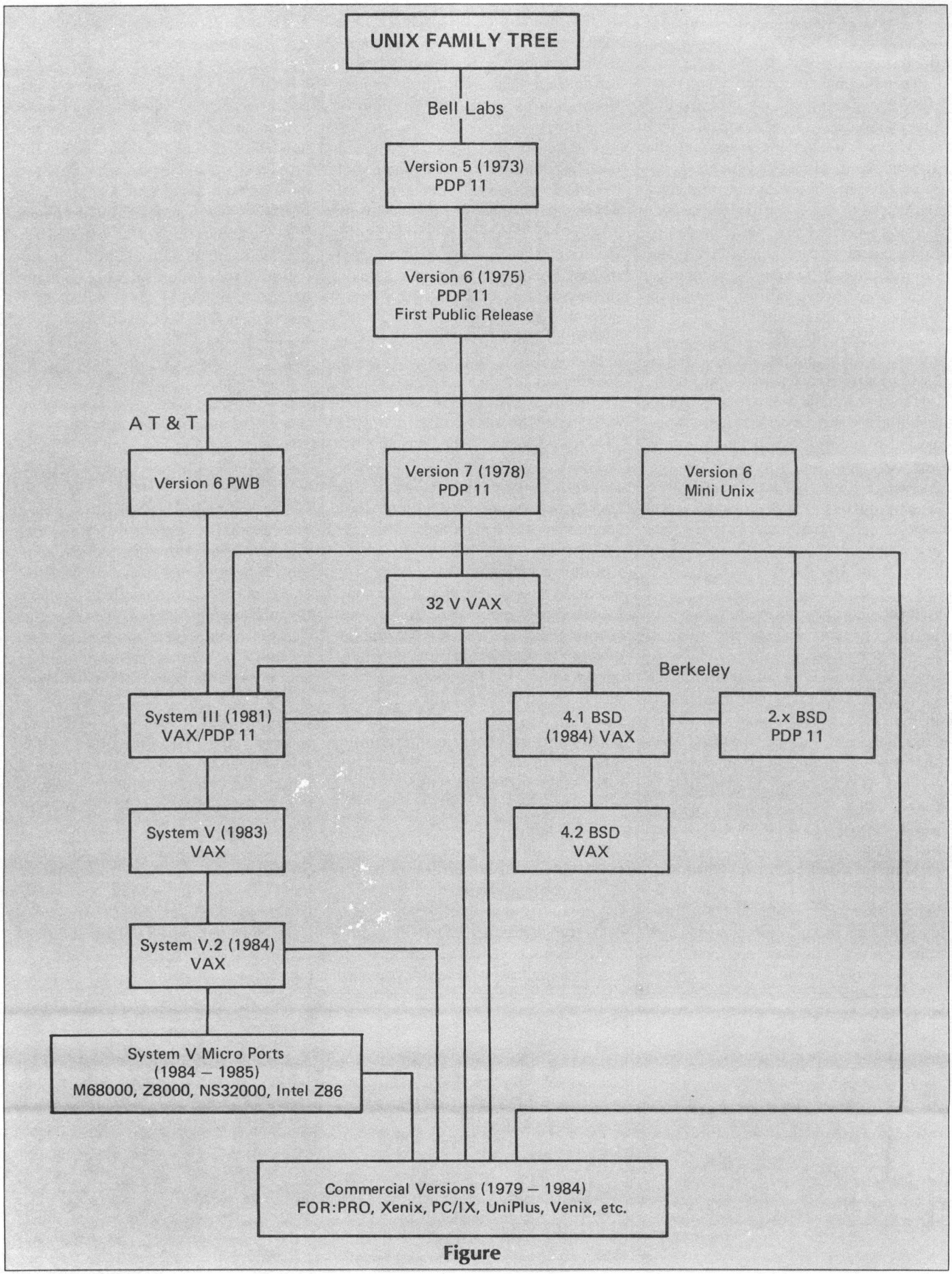
AT&T is making a major effort to establish System V as the standard. But just as 4.2 BSD lacks some of System V's capabilities, System V lacks some important 4.2 BSD features. To some extent, we can expect the best features of both 4.2 BSD and System V to be incorporated into future Unix versions. The catch is that there is a limit to how many 4.2 BSD features can be grafted onto System V; addition of System V features to 4.2 BSD is far less problematic. Much depends on what IBM does, and strategists there must be tempted to thwart AT&T by promoting Unixes other than System V. (It is rumored that IBM is considering making 4.2 BSD available for use under VM.[11]) So

Unix version	Version Seven	4.1 BSD	System III	System V	FOR: PRO	Xenix
Number of Stanix utilities included that are in all six versions	92	92	92	92	92	92
Number of additional Stanix utilities included (Total number of additional Stanix utilities is 103.)	34	51	74	72	87* (54)	79** (41)
Number of additional non-Stanix utilities included	16	105	79	140	57	49** (22)
Number of Stanix system calls included (Total in Stanix is 49.)	43	41	47	48	44	47** (46)
Number of additional system calls	6	15	1	10	20	23** (15)

* This is a projected number for early 1985. The number in parentheses reflects what is officially available at the time of writing (August 1984).

** This number is for the 3.0 release of Xenix. The number in parentheses reflects the pre-3.0 situation.

Table III.
Comparison of Six Unix Versions


UNIX FAMILY TREE
Bell Labs
Version 5 (1973)
PDP 11
Version 6 (1975)
PDP 11
First Public Release
A T & T
Version 6 PWB
Version 7 (1978)
PDP 11
Version 6
Mini Unix
32 V VAX
Berkeley
System III (1981)
VAX/PDP 11
4.1 BSD
(1984) VAX
2.x BSD
PDP 11
System V (1983)
VAX
4.2 BSD
VAX
System V.2 (1984)
VAX
System V Micro Ports
(1984 – 1985)
M68000, Z8000, NS32000, Intel Z86
Commercial Versions (1979 – 1984)
FOR:PRO, Xenix, PC/IX, UniPlus, Venix, etc.

Figure

the extent to which System V will be accepted as a standard remains in doubt.

Spending megabucks on advertising does not guarantee acceptance of a proposed standard. Micro Unix versions based on Version 7 and System III, such as Xenix, PC/IX, VENIX, and FOR:PRO, are in use at far more sites than System V. According to the *Supermicro* newsletter, "with the Xenix announcement for the AT, IBM has administered a possibly fatal blow to AT&T's expensive effort to establish System V as a standard."[12]

Jean Yates of Yates Ventures was quoted in September as saying, "Within 18 months AT&T will be conforming to the IBM standard—Xenix."[13] Micro columnist John Dvorak recently suggested that "AT&T is not only losing key personnel but also may lose the System III vs. System V battle .... The failure of AT&T to add record locking and virtual memory to System V ... may mean that AT&T will lose control of the direction Unix will take."[14]

It should be pointed out that AT&T is aware of System V's deficiencies and is working to overcome them. In a talk given in April 1984 at the European Unix User Group (EUUG) conference at the University of Nijmegen in Holland, Bell Labs' Larry Crume indicated that demand paging will be incorporated into AT&T's kernel as a configuration option in the near future.

The record-locking issue merits special attention because lack of record locking has often been mentioned as a weakness of Unix. To be sure, until recently there was no official Unix standard record-locking mechanism, and some Unix systems still don't offer record locking. But Unix systems aimed at the business marketplace, such as FOR:PRO, VENIX, and Xenix, must offer record locking because it's needed for efficient shared data base applications such as multi-user accounting systems.

The record-locking scheme developed by John Bass[15] is the de facto standard and, as of the summer of 1984, is an official standard adopted by the /usr/group Standards Committee.[16] In the original version developed for Onyx in 1981 and now in the public domain, the key system call is named locking. A more recent implementation calls the key system call lockf. The lockf version is essentially the same as locking but provides enhancements such as the ability to test whether a region is locked without at the same time locking it. This record-locking scheme is effective and widely used, and AT&T has indicated it will cooperate with the /usr/group Standards Committee's adoption of it as a standard.

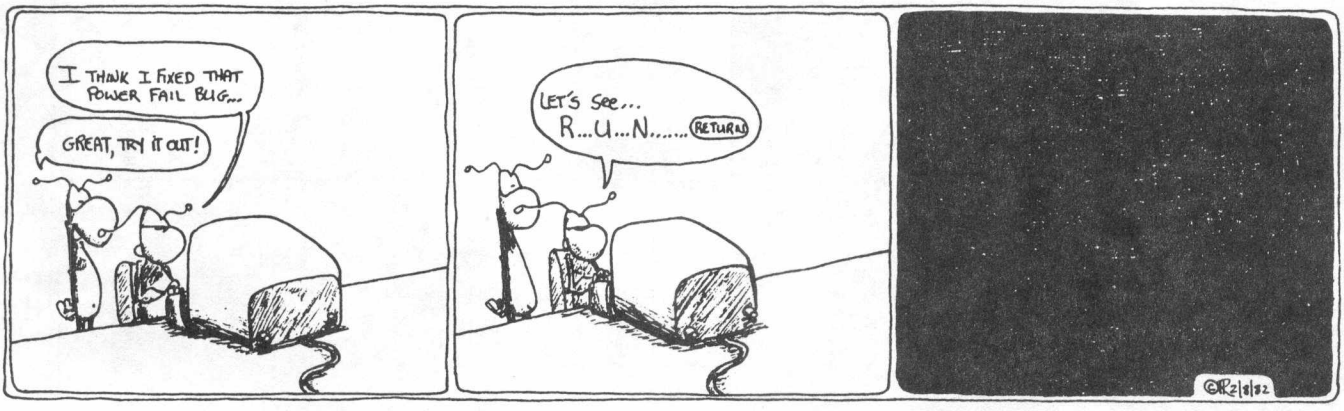
Choosing a Unix

Given the vast variety of Unix versions, which should you select if you are choosing a Unix system? Depending on your circumstances, you should consider not just what is contained in a given version of Unix but such aspects as:

- The likely enhancements in future releases of the version
- How fast the version runs
- How bug-free it is
- The quality of available documentation
- The quality of the hardware that the Unix version runs on
- Availability of support
- The range of software that runs on the version (in particular, whether all the application software you need is currently running in a reasonably well-debugged state on the system)
- Price

Don't decide you need System V just because AT&T has spent so much time and money advertising it. On the other hand, take all the talk about the failure of AT&T to make System V the standard with a grain of salt. As noted earlier, System V's deficiencies are being remedied. System V IPC capabilities provide an effective foundation for integrated application software; this inherent strength, as well as AT&T's marketing effort, will result in continuing integration of System V features, if not true System V, into new microcomputer products. Pundits who proclaim that Xenix on the AT will blow System V out of the water ought to give some serious attention to Microsoft's efforts to produce a new version of Xenix based on System V. Although currently (August 1984) little if any commercially available software makes use of the new System V capabilities, a large quantity of high-quality System V software is likely to be on the market soon.

If the availability of the forthcoming System V software is important to you, bear in mind the possibility of avoiding current System V shortcomings by choosing a Unix that is not certified System V but that offers, or in future releases will offer, System V compatibility. Some microcomputer vendors who advertise System V in reality offer System V features grafted onto another version of Unix, rather than true System V. If it turns out that what is being offered is indeed real System V, be sure to check performance, reliability, and availability of adequate application software before you buy. Of course, these things ought to be



checked before purchasing any other version of Unix, too. The special danger posed by AT&T's System V—and by IBM's products—is that massive marketing efforts, plus the magic of three familiar letters, may lead people to make purchasing decisions without being sufficiently careful to make sure that what they are getting satisfies their requirements.

Notes

[1] This report, titled "The Meaning of Unix," can be obtained by contacting Bill Shattuck at Montgomery Securities (415) 627-2572.

[2] Note, however, that as of August 1984 System V Unix is an exceptional case; there is currently little application software for System V computers.

[3] Information should be sent to Alan Walworth, 825 Clara, Palo Alto, CA 94303. Ideally, I would like man pages so that I can provide a detailed picture of a wide variety of Unix systems. Although ordinary sales literature is better than nothing, it is insufficient for in-depth analysis.

[4] Brian Kernighan, coauthor with Dennis Ritchie of the key reference work *The C Programming Language*, was an early participant in Unix development at Bell Labs. Kernighan coined the term "Unix."

[5] Note that, even if all these problems were taken care of, there would still be a similar problem with options, since you need to know all the options to grasp the full range of issuable commands. To see this point clearly consider that egrep and fgrep could have been implemented as options provided with grep (grep –e and grep –f). Such an implementation would have two fewer commands but no less functionality. So a comprehensive analysis of the capabilities of various Unix versions would require an investigation of just what command options, as well as just what commands, each provides.

[6] Gene Dronek, "PCIX," *UNIX/WORLD*, March/April 1984, p. 37.

[7] However, office-oriented word processing may exist for machines running Xenix. Microsoft and manufacturers of hardware on which Xenix runs should be able to provide information about what is available.

[8] According to Mark S. Zachmann, lex "is missing one of its object libraries." This statement appears in his article "A Venerable UNIX," in *PC*, vol. 3, no. 11, June 12, 1984, pp. 246 – 248.

[9] FOR:PRO also uses a 1024-byte block size. Using a large block size has both pros and cons: to take a simple example, if there are a thousand files, each 400 bytes long, using up a 1024-byte block for each one wastes a lot more disk space than using up a 512-byte block for each (roughly 1000 times 600, or 600,000 bytes wasted versus about 100,000 bytes). 4.2 BSD overcomes this drawback by using blocks 4096 bytes long for most files but arranging, when suitable, to let several small files share a single 4K block. The trade-off is that this and other file system optimization techniques add considerably to the size of 4.2 BSD.

[10] For a good introductory discussion of semaphores, see *Operating System Design: The XINU Approach* by Douglas Comer (Prentice Hall, 1984).

[11] Sol Libes, "News and Views," *Microsystems*, September 1984, p. 8.

[12] *Supermicro*, August 31, 1984 (published by ITOM International Co., P.O. Box 1415, Los Altos, CA 94022).

[13] Jean Yates, *Fortune*, September 17, 1984, p. 70.

[14] John Dvorak, *InfoWorld*, September 10, 1984, p. 96.

[15] John Bass, a Unix expert who assisted with development of Fortune's FOR:PRO, is currently a consultant with DMS Design.

[16] The /usr/group standards are published as a document titled *Proposed Standard: 1984 /usr/group Standard*, available from /usr/group, 4655 Old Ironsides Drive, Suite 200, Santa Clara, CA 95050. The *Reviewer's Guide to the Proposed /usr/group Standard*, which is available with the *Proposed Standard*, includes a listing showing which system calls and subroutines are provided in the /usr/group standard, System V, System III, Version 7, and 4.1 BSD. It also contains some useful background information, an explanation by John Bass of the lockf record-locking standard, and a report by D. Cragun and D. Kretsch of Bell Labs on the ways System V differs from the /usr/group standard. Cragun and Kretsch start off with a peculiar piece of reasoning that suggests AT&T is not wholly enthusiastic about the /usr/group standard: "The current membership of the /usr/group Standards Committee represent end users of Unix systems, developers of Unix systems, developers of systems that are based on or look like Unix systems, and applications developers. Therefore, the /usr/group standard is not a Unix system standard." /usr/group also publishes a *Unix Software Catalog*, a new edition of which is due out in early 1985; this could be a useful starting point for obtaining information on microcomputer Unixes omitted from this article. **DDJ**

Unix Device Drivers

by John L. Bass

Some Unix systems currently being shipped use disk queuing algorithms that optimize throughput. Their performance looks less attractive when you realize that the best-throughput algorithms yield the worst values for response time and fairness.

This article is an introduction to the Unix I/O subsystem and Unix device drivers. The concepts herein cover a wide range of Unix operating systems as well as a number of non-Unix systems. Unix Version 7 drivers are the baseline for the information in this discussion. Device drivers in both earlier and later versions of Unix are similar in concept to these Version 7 drivers but have different system interfaces, queuing structures, and control flow.

For more information on Unix, see *The Bell System Technical Journal*, Part 2, July – August 1978, vol. 57, no. 6. See also "The Unix I/O System" by Dennis M. Ritchie, found in the *Unix Programmer's Manual*, seventh edition, vol. 2B, and in other documents for various releases of Unix.

A Unix I/O Subsystem Overview

Before we look at Unix drivers, we need to take an overview of the Unix kernel to understand the overall system control flow. Shown in Figure 1 (page 40) is the block structure of the Unix I/O system—from a functional view. Each rectangular block in the diagram implements a certain functionality in a modular fashion. Communication between blocks is accomplished via subroutine calls and data structures passed as arguments in those calls. Each rectangular block with rounded corners represents a switch table structure used to select one of several similar functions.

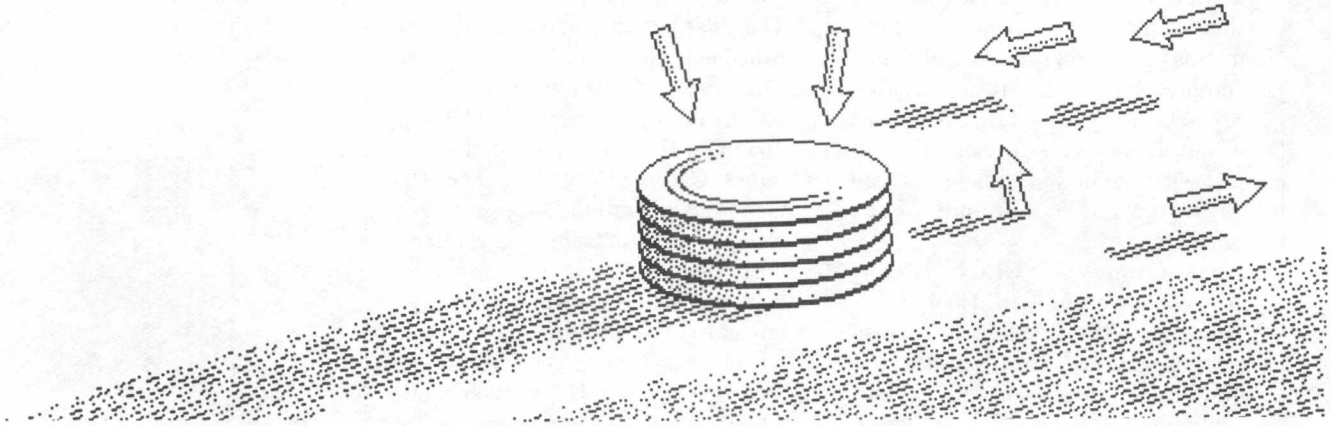
The System Call Interface and Switch

System and application processes interface to the I/O subsystem via the library subroutines open, close, read, write, ioctl, and lseek. Each of these library subroutines copies its arguments according to system call conventions, sets the system call number, and then causes a hardware/software call to the system call manager. This procedure varies from system to system.

The system call manager copies the arguments out of the process' memory and into the kernel's memory. It then uses the system call number to index into the system call switch table (a jump table) to find the proper kernel subroutine to call. The arguments are assembled for the routine, and the routine is called.

When the routine returns, the system call manager copies the return status into the process' memory and resumes execution of the process.

John Bass, 6946 Blue Hill Drive, San Jose, CA 95129.



The File I/O Subsystem

The file I/O subsystem has entry points in the system call switch table for each file I/O system call. The following are the basic entry points used by applications:

Open: scans the directory structure to locate the file named by the application. The inode for the file is brought into memory for use by the other system calls. The inode describes all of the file type, ownership, and allocation information for a file.

If the inode describes a character or block special file, then the driver must be passed the open call. The major device number in the inode is used as an index into either the block or character device driver switch table to obtain the driver's open routine address. If no driver open routine address is found in the switch table, then the driver does not require an open call. If an address is found, then the open routine is called.

If an error occurred, then an error status is returned; otherwise, a file descriptor is returned to the application process.

Close: If the inode is for either a character or block special file and no other process has the inode open, then the close routine (if any) in the switch table is found and the driver's close routine is called.

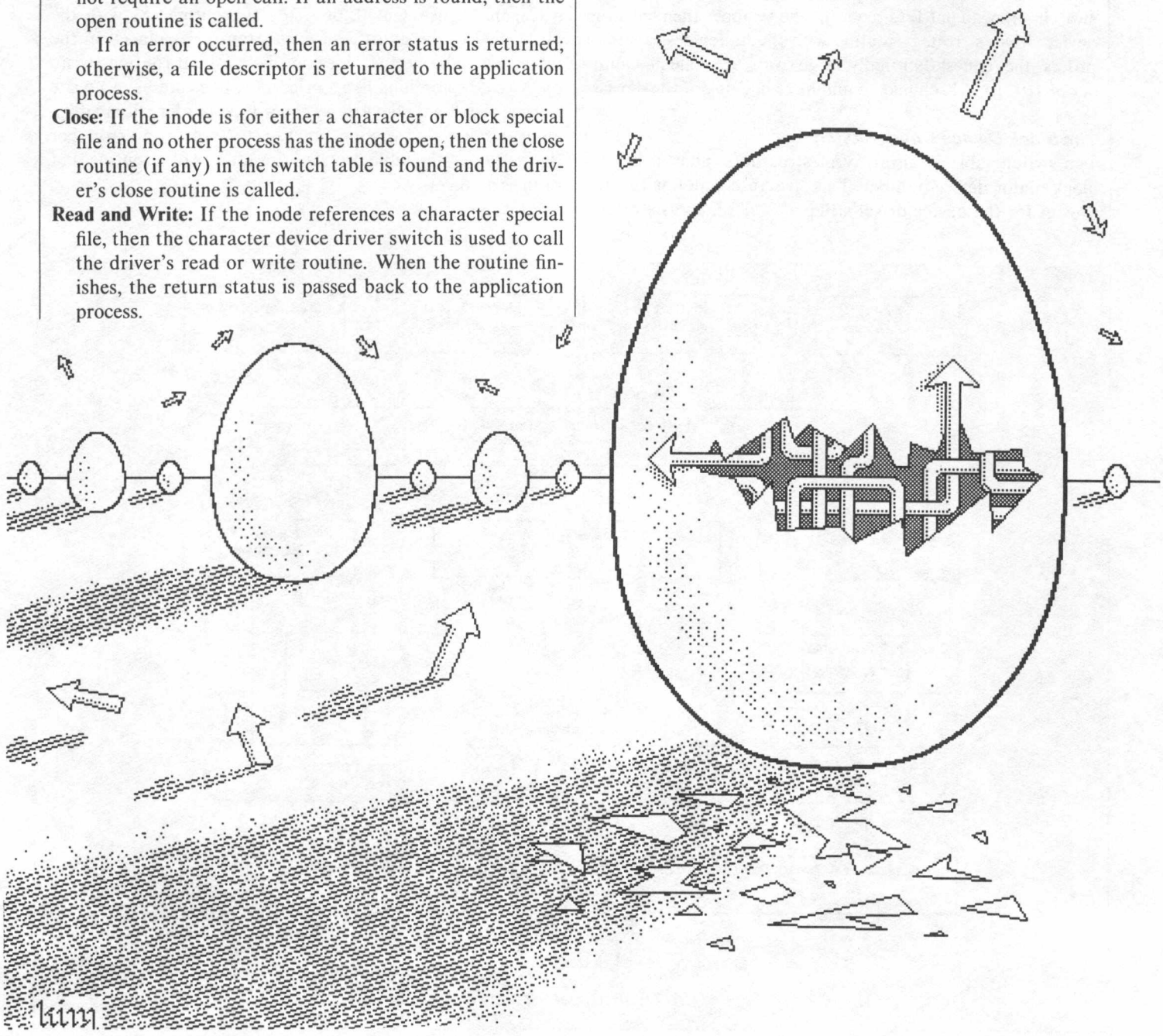
Read and Write: If the inode references a character special file, then the character device driver switch is used to call the driver's read or write routine. When the routine finishes, the return status is passed back to the application process.

If the inode references a block special file, then the current file pointer offset is translated into a logical block number for the device.

If the inode references a file or directory, the current file pointer offset is translated into a logical block in the file. The allocation information for the file is then used to locate the logical block number on the device.

The block I/O subsystem is then requested to return a system buffer with the device data. If it is a read system call, then the data in the buffer is copied into the applications buffer. If it is a write system call, then the data in the applications buffer is copied into the system buffer, and the buffer is flagged to be written back to the device.

This process is repeated for as much data as was requested by the applications system call.



Lseek: This system call does not cause any I/O action to be performed. It simply changes the current file pointer offset used by the read and write routines.

Ioctl: If the inode does not reference a character special file, then an error status is returned; otherwise, the ioctl routine in the device driver is called.

The Block I/O Subsystem

The block I/O subsystem maintains an in-memory buffer pool that forms a simple FIFO cache of blocks read or written. Requested blocks are returned immediately when they are found in the pool. If these blocks are not found in the pool, then the oldest buffer is freed, allocated for this device/block, and then passed to the proper device driver strategy routine.

The Process and Swap Scheduler

The system swapper uses a special buffer descriptor to build a swap in or swap out I/O request. The swapper then calls the device driver's strategy routine with the buffer descriptor to process the request. Normally, large swaps are done in a number of 10K to 60K chunks to minimize hogging of the device.

The Block Device Driver Switch

The switch table is an array of structures indexed by the block major device number. This structure contains the addresses for the device driver's Bopen, Bclose, and Bstrategy subroutines. It also contains the address of the device's I/O queue for use by system metrics. Device drivers are not required to have a Bopen or Bclose routine. When they do not, the entry point for a null subroutine is placed in the table for the missing subroutine.

Each block device driver will have one or more entries in this table. The entries are created during system generation or, on some systems, during autoconfiguration when the system is started up.

The Character Device Driver Switch

The switch table is an array of structures indexed by character major device number. This structure contains the addresses for the device driver's Copen, Cclose, Cread, Cwrite, Cioctl, and Cstop subroutines. Also often included is a pointer to the first TTY structure controlled by the driver for use with system metrics. Device drivers are not required to have a Copen, Cclose, Cioctl, or Cstop subroutine. As with the block device, the address of a null routine is placed in the table. For a device that does not have a read routine (write only—e.g., something like a printer) or does not have a write routine (read only device—a paper tape reader), it is advisable to cause the missing function to return an error. For these devices, the table contains the address of a routine that returns an error.

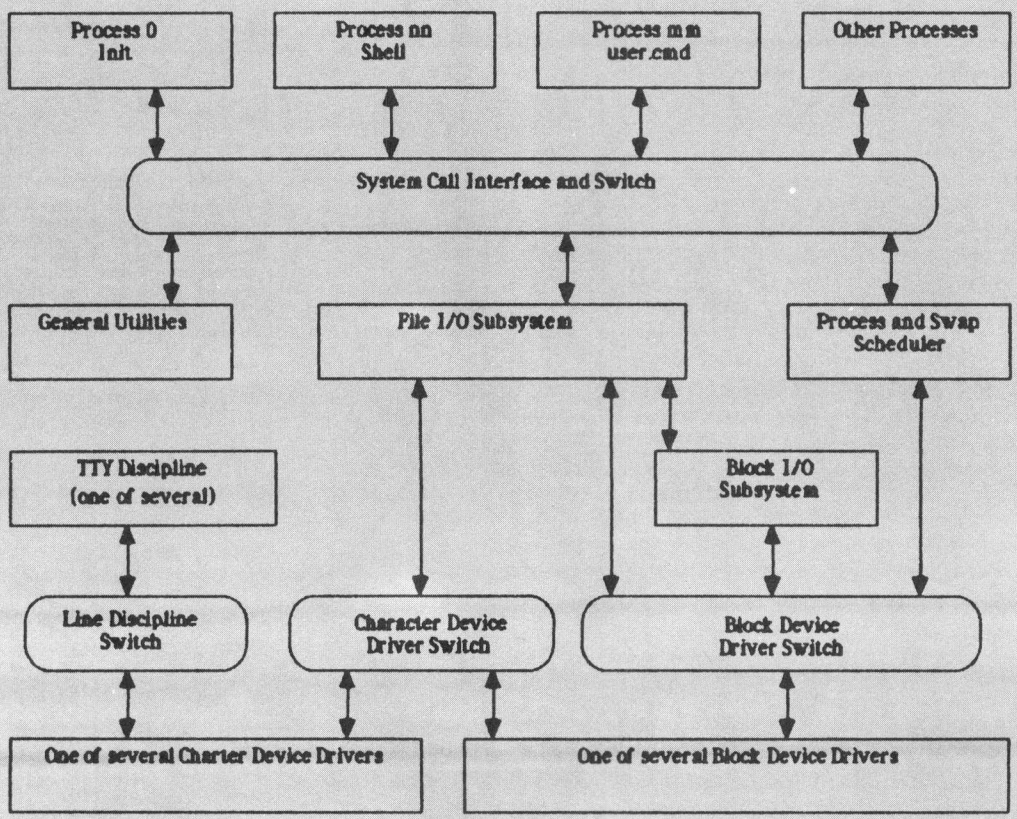


Figure 1

UNIX Functional Structure

The Block Device Driver

A generic disk driver represents the general form of most block device drivers. The structure of a disk driver is given in Figure 2 (page 41).

The block device drivers are used mainly by the file I/O subsystem and block I/O subsystem to provide buffered access to file system devices. In practical terms, these "devices" can be disk drive devices of all kinds, disk emulators (for instance, a RAM disk), and sometimes a network-provided remote disk service.

Most block device drivers also have a simple character driver interface as well. This interface allows direct multiblock transfer to and from the device without using system buffers. This process is mainly used by file system maintenance utilities. The character driver interface also allows the driver to use ioctl calls for special applications (for example, a Format Diskette command).

The Character Device Driver

Character device drivers are used to interface all types of

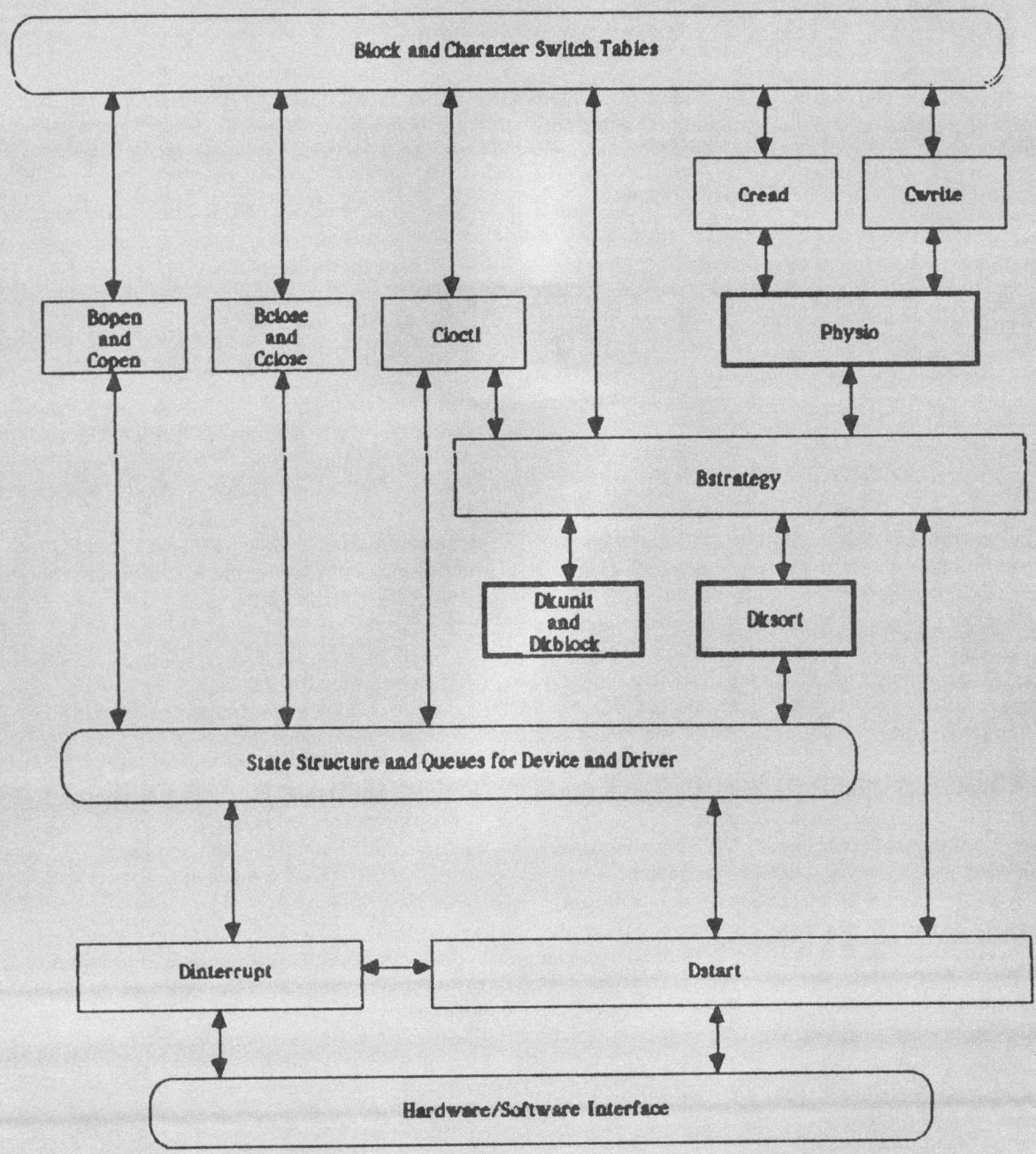


Figure 2

Disk Device Driver

devices to Unix. The driver interfaces the external device to open, read, write, and close system calls to maintain the device-independent I/O model.

An important specialized form of character device driver exists for communication devices that interface terminals to the system. All the special processing to handle echo, character erase, line erase, end of file, xon/xoff flow control, and signal generation is contained in the default tty line discipline. The line discipline switch is an interface to link the default tty line discipline into the terminal driver as well as provide for alternate tty protocols and special tty handling.

Much of the difference between Version 7, Berkeley releases, System 3, and System 5 revolves around changes and improvements in the tty line disciplines. There are significant differences between the implementations for each of these releases.

Figure 3 (page 46) shows the basic outline for a simple terminal driver. This driver completely defaults to using only the default tty line discipline. In practice, a fully implemented driver is much more complex.

The Block Device Driver

This section will discuss the generic disk driver in more detail (see Figure 2). Simple algorithms will be presented for each major routine found in the basic disk driver. The outlines provided here are simplified to some extent. The details vary slightly between various versions of Unix. If you have access to a driver source for your system, consult those listings for more details.

Subroutines Bopen and Copen

The basic function of an open routine is to provide initialization for the controller and drive. Both the block and character open routines may be active at the same time if a sleep is done during initialization of the controller or drive. Likewise, it is possible to have a concurrent close on one interface and an open on the other. Thus, if either open or close sleeps on some I/O, it may be necessary to set some semaphore to prevent race conditions during opens and closes.

There are separate open routines in both the block and character interfaces. To be certain that no minor device is open, the routine must check that no block minor device is open *and* that no character minor device is open. To forget this is a common mistake: it results in the controller/driver getting initialized while active I/O is in progress.

The open routine is called for each open system call for the minor device. The kernel should call open for the root, pipe and swap devices . . . but this is sometimes overlooked or difficult to do.

```
Begin subroutine open(minor_device)
    While open_close_lock set, sleep on open_close_lock
    Set open_close_lock
    If no minor device on this controller is open, then
        Initialize the controller
        Mark controller as ready
    End if
    If no minor device on this drive is open, then
        Initialize the drive
        Mark drive as ready
    End if
    Mark minor_device as open
    Free open_close_lock
End subroutine open
```

Subroutines Bclose and Cclose

This is simply the reverse problem from open with the additional task of flushing and waiting for outstanding buffered writes.

```
Begin subroutine close(minor_device)
    While open_close_lock set, sleep on open_close_lock
    Set open_close_lock
    If any buffered writes for this minor_device, then
        Flag and queue all buffers found
        Wait for minor_devices queue to become empty
    End if
    Clear minor_device open flag
    If no minor device on this drive open, then
        Take drive off line (if required)
        Release drive from controller
    End if
    If no minor device on this controller open, then
        Release controller
        Flag controller as idle
    End if
    Free open_close_lock
End subroutine close
```

Subroutine Cioctl

Most disk drivers do not have an ioctl interface. In those that do, the most common use of the interface is to provide a means for formatting floppy disks (or other removable media).

```
Begin subroutine cioctl(minor_device, cmd)
    If user is not root or cmd is not format, then
        Return error status
    End if
    While raw_buffer is busy, sleep
    Mark raw_buffer as busy
    Set up command parameters in raw_buffer
    Set command type to format
    Call strategy to queue I/O request
    While raw_buffer is not done, sleep
    Mark raw_buffer as free
End subroutine cioctl
```

Subroutine Cread

The normal raw disk read function simply calls physio to validate the user's request, acquire the buffer, initialize the buffer parameters, queue the request, and monitor the I/O completion.

```
Begin subroutine cread(minor_device)
    Call physio to build raw read request for strategy
End subroutine cread
```

Subroutine Cwrite

The normal raw write request is just like the read request.

```
Begin subroutine cwrite
    Call physio to build raw write request for strategy
End subroutine cwrite
```

System Function Physio

The system function physio does all the normal handling to set up a dma transfer to/from an applications buffer. This includes validation of the addresses and lengths requested by the application. The physical addresses are calculated and plugged into the provided buffer header. Physio then calls the provided strategy routine and waits for I/O to be completed. Any error processing is done, and the resulting status is returned to the application.

Subroutine Bstrategy

The strategy routine's job is to take the requested buffer header, map it to the correct device and physical block, and enter it into the queue.

```
Begin subroutine strategy(buf)
    If the requested I/O is outside this partition, then
        Set error flags
        Mark buffer as done
        Return to caller
    End if
    Map correct device and block number from buffer
        parameters using partition table, dkunit, and
        dkblock
    Map block number into cylinder group
    Call dksort to sort request into proper place in queue
    Call Dstart if the device is idle
End subroutine strategy
```

System Functions Dkunit, Dkblock, and Dksort

These system functions are used to implement system standard device interleaving and request queue sorting.

The device interleaving is used to interleave a logical partition over two or more physical drives to balance I/O request loading on the drives. A secondary use of this function is to combine two drives into a larger, single logical partition.

The request queue sorting is used to provide a standard queue optimization. The standard algorithm used is up-elevator (C-Scan) with read preference. This algorithm gives better I/O throughput than FIFO, but allows significant unfairness to be introduced between multiple disk-bound processes. Although this is a good general algorithm, other algorithms produce fairer access to the disk and less erratic response times, while only reducing the total disk throughput by a small margin.

Subroutine Dstart

The purpose of this routine is to translate the parameters in the buffer header into command parameters for the disk controller. The I/O request is then started by the controller. When the request is done, the hardware generates an interrupt, which then calls the Dinterrupt routine. The interrupt routine processes the completion or retry of the last request and calls start to get the disk controller going again.

```
Begin subroutine start
    If request queue is empty, then
        Return
    End if
    Mark device as busy
    If buffer is raw_buffer and command is format, then
        Build format command
        Start format operation
        Return
    End if
    Build transfer command
    Start transfer operation
    Update metrics
End subroutine start
```

Subroutine Dinterrupt

The interrupt first checks for error on the transfer. If an error is found and there have not been too many retries, then the last request is restarted or marked as a hard error.

If the transfer was correct, then the buffer is marked done and start is called for the next request.

```
Begin subroutine interrupt
    If device was idle, then
        Log false interrupt if necessary
        Return
    End if
    If a device error, then
        If a soft error and only a few retries, then
            Clear error condition, if necessary
            Restart controller
            Return
        End if
        Flag error for request
    End if
    Mark request as done
    If more requests in queue, call start
End subroutine interrupt
```

The Simple Serial Terminal Driver

The basic form of a single line terminal driver is shown in Figure 3 (page 46). This driver is the minimal form for a single-character-at-a-time UART with interrupts handled in C code. In practice the Version 7 drivers are significantly more complex, particularly after performance optimization using software pseudo-dma. System 3 and System 5 drivers have additional complexity because device-specific output processing from the tty generic routines is transferred to the driver. It can be expected that even later versions will continue this trend to better support remote network terminals, multi-windowed terminals, and bit-mapped graphics terminals.

Subroutine Copen

```
Begin subroutine copen(minor_dev)
    If not a legal minor_device, then
        Post an error
        Return
    End if
    Initialize tty queuing/state structure
    If first open on device, then
        Set open flag
        Set default initial state flags
        Call system function ttychars for initialization
    End if
    Initialize hardware interface and interrupts
    Call system function ttyopen with queuing structure to
        complete open sequence
End subroutine
```

System Function Ttyopen

The ttyopen function does system-specific initialization that is common to all terminal drivers. This function generally includes establishing terminal ownership, process groups, and marking the queuing structure as open.

Subroutine Cclose

```
Begin subroutine cclose
    Acquire queuing structure for this minor device
    Call system function ttyclose with queuing structure
End subroutine
```

System Function Ttyclose

The ttyclose function performs the common close processing for terminal devices. This generally involves disassociating the device from any process group, waiting for characters in the output queue to finish transmitting, clearing the input queues, and marking the queue as closed.

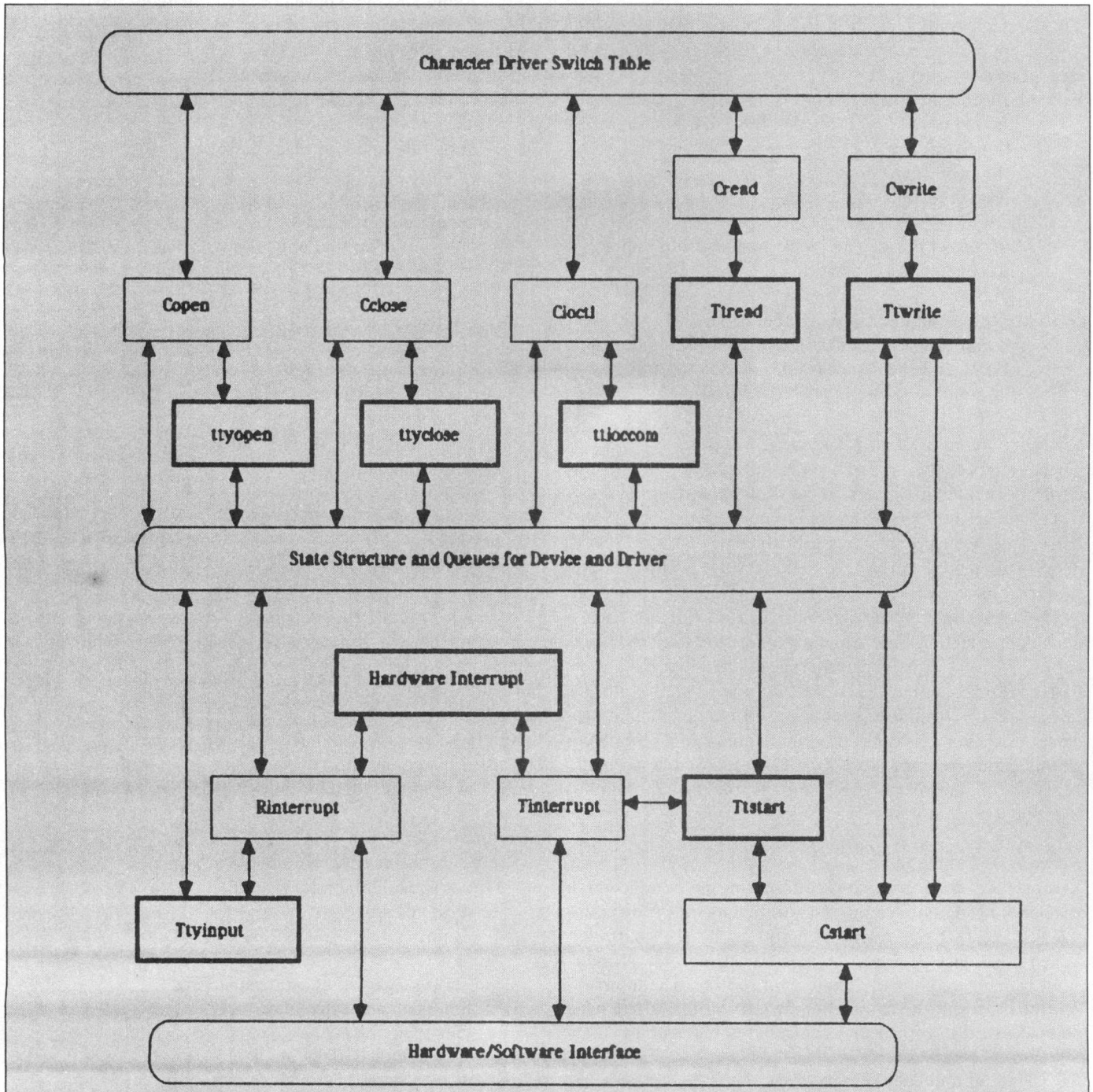


Figure 3

Simple Terminal Device Driver

Subroutine Cioctl

```
Begin subroutine cioctl(minor_device, cmd)
    Acquire queuing structure for this device
    Call system function ttioccom with queuing structure
        and cmd arguments
    If hardware parameters have changed for device, then
        Reprogram hardware to conform to new
            parameters
    End if
End subroutine
```

System Function Ttioccom

The ttioccom function performs the common processing to handle terminal ioctls reading and editing the hardware and software state variables for a serial port.

Subroutine Cread

```
Begin subroutine cread
    Acquire queuing structure for this minor device
    Call system function ttread with the queuing structure
End subroutine cread
```

System Function Ttread

The ttread function calls other routines to process the raw characters on the input queue. Any required editing is done, and the resulting input data is copied into the applications buffer. If more data is required and the input queue is empty, the routine sleeps while waiting for input.

Subroutine Cwrite

```
Begin subroutine cwrite
    Acquire queuing structure for this minor device
    Call system function ttwrite with queueing structure
End subroutine cwrite
```

System Function Ttwrite

The ttwrite function calls other routines to process the outgoing data. Terminal-specific handling is done for tabs, new lines, upper-case and lower-case characters, and delay processing. The routine sleeps as required on the output queue, copies data from the applications buffer to the output queue, and calls the start routine to keep the device busy.

Subroutine Cstart

```
Begin subroutine cstart
    If hardware is not ready, then
        Return
    End if
    If a character is on output queue, then
        Remove character from queue
        If the character is data, then
            Send the character out the serial line
        Else
            Do delay processing
        End if
    End if
End subroutine cstart
```

System Function Hardware Interrupt

By some hardware and/or software means the communications device presents an interrupt request to the system. Often special processing is required to save the hardware/software state at the time of the interrupt. After this state is saved, the proper driver interrupt processing routine is called.

Subroutine Tinterrupt

```
Begin subroutine tinterrupt
    Acquire queuing structure for device
    Call system function ttstart to do any special handling
        and then call driver's cstart routine
    If the output queue has enough free space, then
        Wake up any sleeping applications to fill it up
    End if
End subroutine
```

System Function Ttstart

The ttstart function does common processing before calling the device's start routine. The function generally includes checks to prevent starting a line that is stopped for some reason or is already busy.

Subroutine Rinterrupt

```
Begin subroutine rinterrupt
    Acquire queuing structure for device
    If there is data in receiver, then
        Get character from serial receiver
        Call system function ttyinput to process character
    End if
    Reset interrupt enable as required
End subroutine
```

System Function Ttyinput

The ttyinput function handles common processing of input characters at interrupt time. This generally includes queuing character echo, handling xon/xoff flow control, signals, end of file, and converting input case. The input is queued for edit processing by ttread. A wakeup of ttread is done as required—either on a special condition or on every character if in raw or cbreak mode.

Background

There are many pitfalls for device driver writers. To explore them in reasonable detail would require a three- to four-day seminar. Two interesting problems are common "mis"-features in many Unix systems currently being shipped.

Better Dsort Algorithms

To evaluate various candidate algorithms, readers must first abandon prior knowledge of what is best and then take a good look at the Unix environment and applications.

(1) A large number of studies have been done on disk queuing strategies for batch environments to optimize throughput. Response time and fairness are not addressed as key variables in those studies. Unfortunately, the optimal solutions for throughput have the worst response time and fairness qualities. Computer scientists as a whole tend not to reevaluate old trade-offs in the face of new requirements. The educational community continues to close the minds of newer computer scientists by teaching "truths" that are out of perspective with today's problems.

(2) Queuing algorithms that change the order of writes going to the disk prevent applications from using fail-safe updates to large data bases. With an unknown write ordering, it is impossible to back out certain update transactions after a system crash. FIFO handling of writes is a must.
(3) The standard Unix dsort gives preference to reads and defers all writes to improve response times. This is generally the best policy to prevent blocking on buffered output data. Unfortunately, the standard implementation also defers writes for swapping and paging as well. This ties up the swap and paging buffer headers for long times when several active readers can lock down the request queue. This results in step reductions or nonlinear decay in both system throughput and system efficiency as the disk queue length increases.
(4) Any algorithm that allows one or more processes to dominate the queue is bound to cause a wide variance in response times. Since slow response times are subjective (that is, the user remembers only the worst), then algorithms that generate wide variances are likely to create the "slow response time" conditions that the user will see and remember at random times. If the variance gets worse as the load increases, then the load will become more visible to the user than would normally be expected.

With these requirements in mind it is clear that the standard dsort is nonoptimal for a timesharing system. The standard up-elevator algorithm with deferred writes allows a single active reader of a sequentially allocated file to lock up the disk queue with readahead. This will continue for a maximum scheduling quantum of several seconds. Several such users can create service time delays to other processes in the disk queue of several times the scheduling quantum! If the system is swapping or paging at the same time, then swap and page requests are likely to be serviced once per few seconds—this allows the incore process hogging the queue to do so for even longer periods of time.

The typical up-down elevator algorithm is even worse. The requests for I/O at the ends of the service area get serviced only once per pass, while the middle gets serviced twice per pass. The result is a normal distribution for the probability of service and a process throughput as a function of the location of the file. Processes accessing data at the ends of the request area typically get between 5 – 50% of the normal throughput depending on queue length and access patterns. Note that for most Unix systems the swap/paging areas and the inodes/directories tend to be at one end of the service pattern. The typical result is exec/fork operations that are significantly degraded as the queue length increases, resulting in abnormal response times for simple commands. When swap/paging traffic is affected by this, the system suffers a large step reduction in efficiency and throughput as the swap/paging time exceeds the scheduling quantum.

To fix this behavior requires that lockdown be bounded to short bursts (some lockdown is desired to obtain higher throughputs), and that swapping or paging always get premium service.

The proper algorithm is one that sorts the disk queue in the reverse direction from that in which file data is sequentially allocated. This is down-elevator on most Unix systems. Second, writes on minor devices with file systems are queued FIFO and serviced after all outstanding read requests and swap/page requests have been serviced.

A better variation of this is to service small batches of writes that fall into the natural down-elevator algorithm. Thus, on every pass a few writes are taken off the queue and released. This helps prevent the write-deferred buffer from clogging up the buffer pool. The writes are still done FIFO but do not cause a burst of nonoptimal FIFO traffic in some random access pattern.

Better Serial Interrupt Handling

Character-at-a-time, interrupt-driven servicing can use up large amounts of high-priority, nonschedulable cpu time. For most systems, the time to service an input keystroke in raw mode is between 800 and 3500 microseconds. This includes the time required to process that hardware interrupt, save system state, call the high-level receiver interrupt routine, echo the character, start the output, wake up the sleeping application, and do a context switch to run the process. Not included is the additional 2500 to 4000 microseconds that it takes to perform the read system call and schedulable kernel-level handling of the input data. At 2400 – 4800 baud, most systems are totally cpu bound and will lose incoming data. Normally humans don't type that fast, but repeat functions and multicharacter function keys do!

A common solution to this problem is to handle the receiver and transmit interrupts in a short assembly language interrupt routine. This routine uses a 50 – 200-character dedicated output buffer and a shared 300 – 2000-character input buffer. The tightly coded output interrupt routine requires 30 – 90 microseconds per character, and the normal output routines require 10 – 200 microseconds per character to keep the output buffer full. Similar numbers are true for input. Wakeups are done only on buffer full or empty. A watchdog timeout routine calls the input processor every few clock ticks to process incoming data when the buffer doesn't get full. The net effect is a 70 – 95% reduction in cpu time for serial communications traffic and significantly reduced chances of dropping input characters.

DDJ

A Unix Internals Bibliography

by John Rogers

"You are in a dark, musty, source code directory, with many mysteriously named source files around you . . ."

A number of books and manuals describe how to use the Unix operating system. However, Unix internals documentation is pretty scattered (where it exists at all). This is unfortunate, especially for those of us who work with Unix—the lack of design documents makes our jobs harder. Some public domain Unix-like operating systems (GNU and XINU) are in the works for the rest of the world, and a number of worthwhile ideas in Unix are adaptable to other systems. In the next few years, documentation on the Unix algorithms and data structures will become more valuable to many people than the actual source code licenses for Unix.

Most Unix gurus will tell you that, because Unix is small, written in a high-level language, and available with the source code, documentation isn't as necessary as it is for some other systems. However, Unix has enough code to make earning the title "Unix guru" quite a task. For the first year or two that I was working with Unix, I kept thinking "there has to be some sort of design document that Bell Labs hasn't released," but I haven't found it yet. The normal means of discovering how something works is to "grep for it" in the source code (in fact, there are t-shirts that say "grep for it!"). This isn't always desirable, however, because some of us don't have access to the source code, or, if we do, the source isn't particularly lucid. A comment in the code even says, "You aren't supposed to understand this"!

I've since given up on finding one comprehensive document, but I've become aware that bits and pieces of design documentation are available. Because most of this documentation is in fairly obscure places, I've put together a bibliography to help other people find the information they need.

Using this Bibliography

I have divided the main body of this article into seven subject areas: the kernel, the shell, compilers and language development tools, communications, the C runtime library, miscellaneous utilities, and regular expressions. This is followed by a list of references in alphabetical order by author. Although the organization may seem a little arbitrary, I don't think people will have problems finding what they want.

Within the main body of this article, books, papers, research reports, magazine articles, and the like are cited with the author's name in square brackets; complete bibliographical information appears in the list of references. In the few places where man page references appear, I use the standard form (e.g., LS(1) refers to the man page for "ls" in section 1 of volume 1 of the manual). I have adopted the numbering scheme used in almost every major release except System V (i.e., section 4 has device

You* are *supposed to understand this.

John Rogers, 1740 California Street #6, Mountain View, CA 94041.

Copyright (c) 1984 by John Rogers *(jr@foros1.UUCP). All rights reserved. Unix is a trademark of AT&T Bell Laboratories.*

drivers, and section 5 has file formats); most other things are unchanged.

K&R, of course, refers to *The C Programming Language* by Kernighan and Ritchie (see [Kernighan and Ritchie78a] in the list of references). *UPM* is short for the *Unix Programmer's Manual*, which almost always comes in two volumes: the first contains the man pages, and the second contains a number of papers about Unix (some of which are referenced later). *BSTJ* is the *Bell System Technical Journal* (now the *AT&T Bell Laboratories Technical Journal*), available from Room 1J319, 101 J. F. Kennedy Parkway, Short Hills, NJ 07078. *CSTR* stands for *Computer Science Technical Report*, available (for free) from Room 2C-213, AT&T Bell Laboratories, Murray Hill, NJ 07974. CSRG is the Computer Science Research Group at U. C.–Berkeley.

The Kernel

A group of people at Purdue University wrote a public domain operating system named XINU (a recursive acronym for "XINU Is Not UNIX"); the source code has been published in [Comer]. Although XINU has a lot of the flavor of Unix (especially since it's written in C), it is not a complete time-sharing system by any means; there are no shells or utilities, for instance. For someone unfamiliar with Unix internals, this is a good place to get acquainted with the concepts used in the Unix kernel. For someone who has already read some of the source code to Unix's kernel, however, the XINU code is kind of disappointing.

A document [Lions77] put together for an operating systems course at the University of New South Wales in Australia describes the Unix kernel in some detail. Because it takes the form of annotations to the source code (by line number), it isn't much use without a source listing (see, for instance, [Lions76]). Unfortunately, the document is based on Version 6; there have been four major releases of Unix since then, so it's not exactly current.

[Lions77] presents another problem for people without source licenses. Because it contains a lot of details about the kernel, you're supposed to have a Unix source license to get a copy. Nevertheless, lots of bootleg copies are floating around. For that reason and for the sake of completeness, I'm mentioning this document here. For more information on the whole topic, see [Lions78], which is available to the public.

Allocation of Directory Entries

Allocation of directory entries in most releases of Unix is pretty trivial; 4.2 BSD is the only major exception. See the DIR(5) man page for details on that.

Allocation of Disk Blocks and Inodes

[McKusick, Joy, Leffler, and Fabry] discusses allocation of data blocks and inodes in 4.2 BSD. In other releases of Unix, this allocation tends to be simpler; see the FILSYS(5), FS(5), or FS(4) man page entry for details.

Booting

[Christian] has a pretty good piece (section 19.5) on booting, titled "Booting, Process 0, Process 1." Since booting seems to be different in every release and on every machine, read this with a grain of salt. Section 8 of your Unix manual probably has more detail and accuracy than anything else.

Device Drivers: General

If you have no idea how Unix's device drivers fit in with the rest of the kernel, you might want to read [Comer]; it gives you the flavor of the subject, although a fair number of the details are different from Unix (remember, he's writing about XINU).

Once you have a rough idea of how device drivers fit into Unix (i.e., via major and minor device numbers and the device "switches"), then [Ritchie78] is the thing to read: it discusses the device-switch tables as well as the kernel routines nulldev( ), cpass( ), passc( ), iomove( ), getc( ) and putc( ) (not to be confused with the standard I/O routines of the same names), sleep( ), wakeup( ), timeout( ), spl3( ) and so on, bread( ), getblk( ), breada( ), brelse( ), bwrite( ), bawrite( ), bdwrite( ), geterror( ), and physio( ).

Because [Ritchie78] is terse (like most of Unix), you might want to read something that goes a little slower and gives examples. [McNamara, Vaish, and Bryant], which examines various aspects of device drivers in detail and gives two pages of pseudo-code for a block device driver, discusses sleep( ), wakeup( ), timeout( ), physio( ), spl0-7( ), disksort( ), iowait( ), and deverror( ).

If you're actually going to *write* a device driver, then you may need all the help you can get. [Nystrom] is pretty much the complete authority, with hundreds of pages, lots of source code, and reprints of [Hickman83a] and [Hickman83b]. Unfortunately, you also need a Unix source license, and I'm not sure if that is available other than by taking a seminar from International Technical Seminars (see the list of references).

One last suggestion: According to an article on Usenet by John Levine at INTERACTIVE Systems (john@ima-.UUCP), "the PC/IX manual includes a fairly informative section on writing device drivers." I haven't seen a copy yet.

Device Drivers: Block

[Ritchie78] discusses buffer headers and associated routines. [Hickman83a] is another source of information. As mentioned earlier, [McNamara, Vaish, and Bryant] gives two pages of pseudo-code for a block device driver.

Device Drivers: Character

[Thompson] and [Ritchie78] look at "character lists" (really queues) and the routines that access them.

Directories

Chapter 8 in [Kernighan and Ritchie 78a] discusses the format of directories. In [Kernighan and Pike] (page 59) is the comment that an inode number of zero indicates an empty directory entry.

exec( )

[Johnson and Ritchie 77] mentions that exec( ) builds the argument list for a new program by allocating memory on the stack.

File I/O

[Thompson] in section 4.1 gives some information on the processing done by open( ), including descriptions of the per-user file table, the system file table,

and the inode table; [Christian] in section 19.6 more or less repeats the same information. [Ritchie78] discusses opens on devices. See also namei( ) below—namei( ) is called indirectly when the user does an open( ) or creat( ).

getty

[Christian] in section 19.5 examines the relationships between init, getty, and login, though not in much detail. Note that the parameters passed to getty tend to vary from one implementation of Unix to the next. See the GETTY(8) or GETTY(1M) man page for details.

iget( ) and iput( )

[Wales] discusses iget( ) and iput( ). iget( ) is called to locate or read in an in-core copy of an inode, given its inode number; reference counts are used to keep track of how many processes are using a given inode. iput( ) decrements the reference count and, when the reference count goes to zero, frees the in-core copy, writing it out to disk if it has been modified.

[Wales] also discusses the hashing used to find in-core inodes—and the use of a singly linked list if collisions occur. [Lankford] mentions that, as of System V and 4.1 BSD, inode table entries are hashed (in releases prior to those, the table was searched sequentially), and [Ritchie79] notes that in the original PDP-7 Unix, iget( ) left the inode it found in a constant location.

init

Section 6.6 in [Ritchie and Thompson] gives a superficial overview of how init works (ignoring both getty and login). The INIT(8) (or INIT(1M))man page provides more detail. Of course, init keeps changing from one release to the next. [Gauthier] in section 19.5 supplies a great deal of detail about how process 1 is created "from scratch."

Memory Management

[Johnson and Ritchie 77] includes a short discussion of various aspects of memory management in the Unix kernel. [Thompson] observes that "the swapping process is the only process that waits for primary memory to become available."

mount( )

[Ritchie and Thompson] in section IV and [Thompson] in section 4.2 discuss how the mount table is built and how namei( ) uses it.

namei( )

This routine in the kernel converts from a path name to an inode number. It is mentioned in [Leffler, Karels, and McKusick], along with the fact that 4.2 BSD has namei( ) caching—with a table of names, inode numbers, device numbers, and pointers to inode table entries. This same source explores the logic used by namei( ). Sections 4.1 and 4.2 in [Thompson] discuss various aspects of the translation of the name to the inode number. (See also iget( ) and iput( ) above.)

Process Table

[Leffler, Karels, and McKusick] mentions that the newproc( ) routine in the kernel does a linear search of the process table to allocate an ID for each new process. [Goodwin] describes in significant detail various algorithms for allocating new process IDs.

[Leffler, Karels, and McKusick] also gives a list of other kernel routines and/or system calls that do sequential searches of the process table, as well as the reasons for each search. The routines are: exit( ), wait( ), fork( ), newproc( ), kill( ), gsignal( ), schedcpu( ), and sched( ).

Read-ahead

The Unix kernel implements a read-ahead scheme where the kernel tries to predict which disk blocks will need to be read in and to read them in before they are actually requested by a user program. This is discussed in [Ritchie78]. [Goodwin] proposes an improvement on the algorithm.

Scheduling

[Gauthier] in section 19.3 looks briefly at scheduling, as does [Thompson] in section 2.3. You can glean other details from a careful reading of the PS(1) man page.

u page

Much of the data about a process is stored in the u page (so called because the global variable with its address is named "u"); this is discussed in [Ritchie78] and [Goodwin], where it is called the per process data area, and in [Thompson], where it is called the system data segment. Note that a u page is different from a process table entry (of which there is also one per process): the u page is swapped to disk, while the process table entries are always resident in main memory.

Update Process

[Gauthier] on page 204 mentions the update process (with /etc/update showing in the "ps" example). On page 213 is the comment: "the /etc/update program forces disk updates every thirty seconds and should be run at every installation."

The Shell

[Ritchie and Thompson] in section 6.5 describes the shell's use of exec( ), fork( ), wait( ), and read( ); how it handles pipes and I/O redirection; and how shell scripts work.

Background Processes

[Joy] shows how the C shell handles its table of background processes.

cd

Most man pages for CD(1) (or CHDIR(1)) mention that the command must be built into the shell; [Ritchie79] gives an interesting historical note on how the authors of Unix discovered this.

Filename Expansion

See the section on regular expressions (page 56).

I/O Redirection

[Holt] on page 169 gives a sketchy description of how this is handled.

Running Programs

[Holt] on pages 167-168 presents some pseudo-code for the steps a shell goes through when it creates a subshell and runs a given program; it does not, however, describe how shell scripts are handled when they are run. See also the discussion of the system( ) subroutine in the section on the C runtime library (page 54.)

Compilers and Language Development Tools

cpp

[Johnson and Lesk] mentions that the

C preprocessor is written using yacc and lex. Martin Minow posted a complete public domain cpp to Usenet this summer, and another partial implementation ("p") is given with full source code listings in [Schreiner].

lex

[Johnson and Lesk] states that lex is written using yacc. [Aho and Ullman] has quite a bit of the theory concerning lex in its chapter 3. See also the section on regular expressions. A public domain version of lex is available from Decus.

lint

[Johnson78a] touches on the design of lint in the section on implementation.

pcc

[Johnson and Lesk] mentions that pcc is written using yacc and lex.

[Johnson79] is a long, detailed introduction to the guts of the portable C compiler. It discusses parsing, symbol table handling, expression trees, and register allocation. It gives a couple of examples of the templates used to rewrite expression trees and eventually to generate code, as well as a number of the function and source filenames for the compiler.

[Leffler] is an exhaustive document (100 pages, although some of it applies only to the Harris /6). It examines the organization of the compiler in great detail, giving source file and function names throughout. Surprisingly, it gives very few examples of expression trees and rewriting templates. It is very good nonetheless.

The Ritchie (PDP-11) C Compiler

[Ritchie76] is, of course, the authoritative paper on the PDP-11 C compiler. (Most C compilers are derived from pcc, but there are a few. . . .)

[Pammett] discusses porting the Ritchie C compiler to the TI990. For a paper of this size (200+ pages), it says amazingly little about how the compiler works. An appendix, however, gives a detailed description of the code tables.

yacc

[Johnson78b] examines how the parser built by yacc operates and provides a detailed example of a parse table. The theory behind yacc gets attention in [Aho and Ullman], which contains pointers to other papers and books on the subject as well. This book also presents the grammar and parse table for a subset of eqn.

A public domain implementation of yacc is available from Decus.

Communications

Networking in 4.2 BSD

[Leffler, Joy, and Fabry] devotes 29 pages to the nitty-gritty of sockets and protocol handling in the 4.2 kernel.

UUCP

[Nowitz] discusses queuing, work file formats, control files, and a bunch of the "setup" protocol, but doesn't offer much about the main file transfer protocol (the "g" protocol). [Nowitz and Lesk] looks at the initial handshaking.

For those who are really interested in how the "g" protocol fits into UUCP, Piet Beertema of CWI in Amsterdam (piet@mcvax.UUCP) wrote another protocol—the "f" protocol for use on X.25 networks—that "drops into" UUCP. The source code for this protocol is public domain and was posted to Usenet this summer.

The C Runtime Library

bsearch( )

bsearch( ), the binary search routine from System V, uses algorithm B in section 6.2.1 of [Knuth] (according to the man page).

Calling Sequences

[Johnson and Ritchie 81] discusses these in great detail.

ctype (isalpha( ), isascii( ), etc.)

This is pretty trivial, but a paper was actually published that talks about how the CTYPE(3) macros work; see [Gimpel] if you're interested.

dbm(3)

The algorithm used by the data base routines dbminit( ), etc., which are documented in the DBM(3) man page, is described in [Fagin, et al.], [Carter and Wegman], and possibly in the November 1982 *BSTJ* (part two).

hsearch( ), hcreate( ), hdestroy( )

These hash table routines (in System V) use algorithm D from section 6.4 of [Knuth] (according to the man page).

isatty( )

[Arnold] mentions that isatty( ) simply calls gtty( ) and checks the return value.

lsearch( ), lfind( )

These linear search routines (lfind( ) was added in System V, release 2) use algorithms from section 6.1 of [Knuth].

Memory Allocation

[Kernighan and Ritchie 78a] gives listings of alloc( ) and free( ) in section 8.7. Various algorithms are used in different releases of Unix: System V has two different versions of malloc( ), and 4.2 BSD supposedly has a power-of-two allocation scheme.

printf( ), fprintf( ), and sprintf( )

For source listings of more or less complete printf( ) functions, see [Hendrix] or [Comer].

Some versions of the printf( ) family call an internal routine named _doprnt( ): see, for instance, [Kernighan and Pike], page 189. Eric Kiebler (eric@washu.UUCP) posted this to Usenet a while back:

"BEWARE the _doprnt
code, my son!
The bits that twitch;
the types that clash!
Use the portable
varargs stuff,
Avoid the _doprnt's
teeth that gnash!"

qsort( )

This uses the quicker-sort algorithm, which has been described everywhere including section 5.2.2 of [Knuth]; see especially page 123 where qsort( ) chooses a trial median from the center of the list.

stdio

A great deal of detail on the workings of the standard I/O library is given in chapter 8 of [Kernighan and Ritchie 78a].

Strings (strcmp( ), strtok( ), etc.)

A public domain implementation of the various string functions (literally

dozens of them) was written by Henry Spencer and posted to Usenet.

system( )
Listings of various versions of the system( ) subroutine abound; see [Kernighan and Pike], pages 223-229, and [Kernighan and Ritchie 78b], section 6.

System Calls
Some manuals give the assembly language statements necessary to invoke a given system call (V7 and 4.1 BSD both do this). Later versions of Unix, however, don't do this, presumably because almost no one writes in assembly language anymore. See the section on the kernel (page 51).

tsearch( ), tfind( ), tdelete( ), twalk( )
These routines from System V are for working with binary trees; they use algorithms T and D given in section 6.2.2 of [Knuth].

Miscellaneous Utilities

adb
You can get some hints on how adb does its work from the PTRACE(2) (process trace) man page; note that the details of the system call differ depending on the hardware and the Unix release.

ar
There are various versions of ar (file archiver) floating around. The most recent one (in 4 BSD and System V, Release 2) has a public domain equivalent in the par (portable archive) and unpar programs.

at
[Thomas and Yates] discusses atrun and /usr/spool/at.

awk
[Aho, Kernighan, and Weinberger] has a particularly concise paragraph in section 5 that explains how awk works: it uses yacc and lex; regular expressions are handled by deterministic finite automata; awk builds a parse tree from the program, which is interpreted when actually processing data. [Kernighan and Pike] mentions on page 124 that awk uses hashing to implement its associative arrays. See also the section on regular expressions.

bc
The BC(1) man page mentions that bc is a preprocessor for dc (see below). Various authors note that it is written using yacc.

bdiff
[McGilton and Morgan] on page 173 says that bdiff uses split and diff to do its work.

dc
[Morris and Cherry] discusses certain details of dc, including how it does arithmetic and the dynamic storage allocation method it uses (described in more detail in [Knowlton]).

df
The df (display filesystems) utility calculates the amount of free space on a file system differently in different releases: in V7 it reads the disk's free list, in 4.x BSD it reads the superblock, and in Systems III and V it uses the ustat(2) system call. (This information is from Usenet.)

diff
The algorithm used in diff was developed (independently) by Harold Stone and by Wayne Hunt and Tom Szymanski; see [Hunt and Szymanski] or [McIlroy and Hunt] for details.

echo
Just about every book on C shows how the echo command accesses the command line arguments; [Kernighan and Ritchie 78a] gives three different versions on page 111.

ed
You can get some of the flavor of ed from [Kernighan and Plauger 76] and [Kernighan and Plauger 81]; see also the section on regular expressions.

egrep
egrep uses an extended version of the normal regular expression algorithms; see the section on regular expressions.

eqn
[Kernighan and Cherry] in sections 5 and 6 supplies various details about the design of eqn, including a simplified version of the yacc grammar that it uses.

fgrep
[Whale] gives the source code for a public domain version of fgrep; the comments discuss the Knuth-Morris-Pratt string-matching algorithm.

grep
[Kernighan and Pike] discusses the tradeoffs between the different algorithms used in grep, fgrep, and egrep. See the section on regular expressions.

passwd (command)
[Gauthier] provides on page 196 a short explanation of why the passwd command must be set-uid root (if this isn't already obvious).

pwd
[Kernighan and Pike] gives hints on pages 51-52 about how pwd works.

spell
See [Kernighan and Pike], page 314, [Peterson], and [McIlroy].

Regular Expressions
Regular expressions are used to varying degrees in awk, ed, egrep, Jim Gosling's emacs, grep, lex, rn (a "readnews" replacement that was posted to Usenet this summer), sed, the shells, vi (Berkeley's screen-oriented editor), and yacc. While they're very popular under Unix, they aren't used very often in other environments. Rather than discuss regular expressions under each utility that uses them, I've collected all the information here.

Implementation
[Holub] contains source code in C for a version of grep that implements full regular expressions. [Kernighan and Plauger 76] and [Kernighan and Plauger 81] give source code to handle slightly different regular expressions in RATFOR and Pascal, respectively.

Theory
The standard reference is [Aho and Ullman], which cites various other publications on the subject. [Aho and Corasick] may also be a worthwhile paper to read; I haven't seen a copy yet.

References

[Aho and Corasick] "Efficient String Matching: An Aid to Bibliographic Search," by Alfred V. Aho and M. J. Corasick, in *Communications of the ACM*, 1975, volume 18, pages 333-340.

[Aho, Kernighan, and Weinberger] "Awk–A Pattern Scanning and Processing Language," second edition (September 1978), by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, in *UPM*, volume 2. (The first edition of this paper (July 1978) is available in *Software–Practice and Experience*, volume 9.)

[Aho and Ullman] *Principles of Compiler Design*, by Alfred V. Aho and Jeffrey D. Ullman, Addison-Wesley, 1977.

[Arnold] "Screen Updating and Cursor Movements Optimization: A Library Package," by Kenneth C. R. C. Arnold, in *UPM* for 4.x BSD, volume 2.

[Bach and Buroff] "A Multiprocessor Unix System," by Maurice J. Bach and Steven J. Buroff, in *USENIX 1984 Summer Conference Proceedings*, 1984. (See [USENIX Summer '84].)

[Bourne] *The Unix System*, by Steve R. Bourne, Addison-Wesley, 1982.

[Carter and Wegman] "Universal Classes of Hash Functions," by Carter and Wegman, in *Proceedings of 9th SIGACT* 1977.

[Christian] *The Unix Operating System*, by Kaare Christian, John Wiley and Sons, 1983.

[Comer] *Operating System Design, The XINU Approach*, by Douglas E. Comer, Prentice-Hall, 1984.

[Fagin, et al.] "Extendible Hashing–A Fast Access Method for Dynamic Files," by Fagin, Nievergelt, Pippenger, and Strong, in *Transactions on Database Systems*, 1979, volume 4, number 3.

[Gauthier] *Using the Unix System*, by Richard Gauthier, Reston, 1981.

[Gimpel] "The Minimization of Spatially Multiplexed Character Sets," by James F. Gimpel, in *Communictions of the ACM*, June 1974, volume 17, number 6, pages 315-318.

[Goodwin] "System V Performance Enhancements," by Ken Goodwin, in *;login:*, July 1984, volume 9, number 3, pages 6-11.

[Hendrix] *The Small-C Handbook*, by J. E. Hendrix, Reston, 1984.

[Hickman83a] "The Unix Buffer Mechanism–A Detailed Look," by Kipp E. B. Hickman, 1983. (Reprinted in *Writing Unix Device Drivers*; see [Nystrom].)

[Hickman83b] "The Basics of Writing Unix Device Drivers, with References to the Motorola 68000," by Kipp E. B. Hickman, 1983. (Reprinted in *Writing Unix Device Drivers*; see [Nystrom].)

[Holt] *Concurrent Euclid, the Unix System, and Tunis*, by R. C. Holt, Addison-Wesley, 1983.

[Holub] "GREP.C–A Unix-like, Generalized, Regular Expression Parser," by Allen Holub, in *Dr. Dobb's Journal*, October 1984, number 96.

[Hunt and Szymanski] "A Fast Algorithm for Computing Longest Common Subsequences," by J. W. Hunt and Tom G. Szymanski, in *Communications of the ACM*, May 1977.

[Johnson78a] "LINT–A C Program Checker" (July 26, 1978), by Stephen C. Johnson, in *UPM*, volume 2. (Also available as *CSTR*, number 65.)

[Johnson78b] "YACC: Yet Another Compiler-Compiler" (1978), by Stephen C. Johnson, in *UPM*, volume 2. (Also available as CSTR, number 32.) *Note: the date for this is variously given as 1974, 1975, or 1978; because it contains references to items published in 1978, that seems to be the correct date.*

[Johnson79] "A Tour Through the Portable C Compiler" (1979), by Stephen C. Johnson, in *UPM*, volume 2.

[Johnson and Lesk] "Language Development Tools" (December 27, 1977), by Stephen C. Johnson and Michael E. Lesk, in *BSTJ*, July/August 1978, volume 57, number 6, part 2.

[Johnson and Ritchie 77] "Portability of C Programs and the Unix System" (December 5, 1977), by Stephen C. Johnson and Dennis M. Ritchie, in *BSTJ*, July/August 1978.

[Johnson and Ritchie 81] "The C Language Calling Sequence" (September 1981), by Stephen C. Johnson and Dennis M. Ritchie, *CSTR*, number 102.

[Joy] "An Introduction to the C Shell," by William N. Joy, in *UPM* for 4.x BSD, volume 2.

[Kernighan and Cherry] "A System for Typesetting Mathematics," by Brian W. Kernighan and Lorinda L. Cherry, in *UPM*, volume 2. (An earlier version of this appeared in *Communications of the ACM*, March 1975.)

[Kernighan and Pike] *The Unix Programming Environment*, by Brian W. Kernighan and Rob Pike, Prentice-Hall, 1984.

[Kernighan and Plauger 76] *Software Tools*, by Brian W. Kernighan and P. J. Plauger, Addison-Wesley, 1976.

[Kernighan and Plauger 81] *Software Tools in Pascal*, by Brian W. Kernighan and P. J. Plauger, Addison-Wesley, 1981.

[Kernighan and Ritchie 78a] *The C Programming Language*, Brian Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.

[Kernighan and Ritchie 78b] "Unix Programming–Second Edition" (November 12, 1978), by Brian Kernighan and Dennis M. Ritchie, in *UPM*, volume 2.

[Knowlton] "A Fast Storage Allocator" (October 1965), by K. C. Knowlton, in *Communications of the ACM*, volume 8, pages 623-625.

[Knuth] *The Art of Computer Programming–Volume 3, Sorting and Searching*, by Donald E. Knuth, Addison-Wesley, 1973.

[Lankford] "Unix System V and 4 BSD Performance," by Jeffrey P. Lankford, in *USENIX 1984 Summer Conference Proceedings*, 1984 (See [USENIX Summer'84].)

[Leffler] "A Detailed Tour Through the /6 Portable C Compiler," (1980?), by Samuel J. Leffler, Department of Computer Engineering, Case Western Reserve University.

[Leffler, Joy, and Fabry] "4.2 BSD Networking Implementation Notes" (July 1983), by Samuel J. Leffler, William N. Joy, and Robert S. Fabry, in *UPM* for 4.2 BSD, volume 2C. (Also available as UCB CSRG TR/6.)

[Leffler, Karels, and McKusick] "Measuring and Improving the Performance of 4.2 BSD," by Sam Leffler, Mike Karels, and M. Kirk McKusick, in *USENIX 1984 Sum-*

mer Conference Proceedings, 1984. (See [USENIX Summer'84].)

[Lesk and Schmidt] "LEX—A Lexical Analyzer Generator" (1978), by Michael E. Lesk and Eric Schmidt, in *UPM*, volume 2.

[Lions76] "Unix Operating System Source Code, Level Six" (1976), assembled by John Lions, School of Electrical Engineering, University of New South Wales.

[Lions77] "A Commentary on the Unix Operating System" (1977), by John Lions, School of Electrical Engineering, University of New South Wales.

[Lions78] "An Operating System Case Study" (1978), by John Lions in *Operating Systems Review*, July 1978, volume 12, number 3.

[McGilton and Morgan] *Introducing the Unix System*, by Henry McGilton and Rachel Morgan, McGraw-Hill, 1983.

[McIlroy] "Development of a Spelling List" (1982), by M. Doug McIlroy, in *IEEE Transactions on Communications*, January 1982.

[McIlroy and Hunt] "An Algorithm for Differential File Comparison" (1976), by M. Doug McIlroy and J. Wayne Hunt, *CSTR*, number 40 (41?). (Unfortunately, this is obsolete and no longer available from AT&T.)

[McKusick, Joy, Leffler, and Fabry] "A Fast File System for Unix" (July 27, 1983), by Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, in the *UPM* for 4.2 BSD, volume 2C. (Also available as CSRG TR/7.)

[McNamara, Vaish, and Bryant] "Writing Device Drivers for XENIX Systems," by Jean McNamara, Paresh Vaish, and Richard N. Bryant, in *UniForum Conference Proceedings*, January 1984.

[Morris and Cherry] "DC—An Interactive Desk Calculator" (November 15, 1978), by Robert Morris and Lorinda L. Cherry, in *UPM*, volume 2.

[Nowitz] "*UUCP* Implementation Description" (1978?), by D. A. Nowitz, in *UPM*, volume 2 (of most manuals).

[Nowitz and Lesk] "A Dial-Up Network of Unix Systems" (1978), by D. A. Nowitz and M. E. Lesk, in *UPM*, volume 2 (of most manuals).

[Nystrom] *Writing Unix Device Drivers*, by Bob Nystrom, International Technical Seminars, 1984. (Requires a Unix source license; only available by taking the seminar?)

[Pammett] "A Compiler for the Systems Programming Language—C" (Master's thesis, 1979), by Kevin G. Pammett, Department of Computer Science, University of Waterloo.

[Peterson] "Computer Programs for Detecting and Correcting Spelling Errors" (1980), by James L. Peterson, in *Communications of the ACM*, December 1980, volume 23, number 12.

[Ritchie76] "A Tour through the Unix C Compiler" (1976?), by Dennis M. Ritchie,in *UPM*, volume 2.

[Ritchie78] "The Unix I/O System" (1978), by Dennis M. Ritchie, in *UPM*, volume 2.

[Ritchie79] "The Evolution of the Unix Timesharing System" (September 1979), by Dennis M. Ritchie, presented at the Symposium on Language Design and Programming Methodology in Sidney, Australia, reprinted in *Microsystems*, October 1984, volume 5, number 10.

[Ritchie and Thompson] "The Unix Time-Sharing System" (1978), by Dennis M. Ritchie and Ken Thompson, in *BSTJ*, July/August 1978, and *UPM*, volume 2, of most manuals.

[Schreiner] "p–A Small-C Preprocessor" (1984), by Axel T. Schreiner, in *Dr. Dobb's Journal*, July 1984, number 93.

[Thomas and Yates] *A User Guide to the Unix System*, by Rebecca Thomas and Jean Yates, Osborne/McGraw- Hill, 1982.

[Thompson] "Unix Implementation" (1978), by Ken Thompson, in *BSTJ*, July/August 1978, and *UPM*, volume 2.

[USENIX Summer'84] *USENIX 1984 Summer Conference Proceedings*, USENIX Association and the Software Tools Users Group, June 12-15, 1984, in Salt Lake City.

[Vaish and McNamara] "Techniques for Debugging XENIX Device Drivers" (1984), by Paresh K. Vaish and Jean Marie McNamara, in *USENIX 1984 Summer Conference Proceedings*, 1984. (See [USENIX Summer '84].)

[Wales] "Re: 'panic: iput' message" (October 19, 1983), by Rich Wales (wales@UCLA-LOCUS.ARPA), message number 12791@sri-arpa.UUCP on Usenet.

[Whale] "fgrep.c—new version" (August 8, 1984), by Geoff Whale (goeff@elecvax.OZ), message number 296@elecvax.OZ on Usenet.

DDJ

A File Browser Program

by John R. Johnson

Casting about for a useful problem to exercise my new C compiler, I heard about a useful program on a VAX system called "show." This program is similar to the "type" intrinsic command in CP/M, but it allows you to move about at will in the file being displayed. I have often been irritated by "type" when I am looking for a line near the end of a large file and it scrolls past before I can stop the display. It could be a useful utility.

Browse is essentially the front end of a simple editor program. Since it will not change the file, but only display it, a line orientation is adequate. To make moving around easier, the lines in the display should be numbered.

A simple command line parser is required to make the program useful. The ability to repeat a command a set number of times is a desirable option. The parser should accept numeric arguments for either repeat count or line number, depending on the command function.

Power can be superfluous and features can just be in the way when all you want is the right tool for the task.

There are existing utilities to list files on the printer. A list option in a file browser would be worth including only for those cases where hard copy is desired for only a few lines. If this could be included it would be worth doing.

The most difficult decision involved reading in the file. Should the file be buffered on disk or kept entirely in memory? If it is kept in memory the random access display would be quick. Response time would suffer with a disk buffer. The trade-off is file size. If it is kept entirely in memory the file must be short enough to fit or it will be truncated.

If the file browser has a slow response there is no real reason to use it rather than the editor to examine a file. Any editor will allow most of the functions of the browser. Since it is used primarily for looking over program source code, and I believe programs should be kept as short as possible, I opted for speed. Browse will arbitrarily chop the end off of any file that is too long to fit in the available memory.

The hardware-dependent features are severely restricted. Direct cursor addressing is specifically not required. A string that clears the screen and homes the cursor is used. If your terminal does not support this feature, locate each instance of the #define constant CLEARS, and replace the call puts(CLEARS) with a call to a function to write a screen full of blanks. Edit the file Browse.h to set the screen and printer page sizes to correspond with your hardware, and that should be all of the installation required.

I do not recommend using I/O redirection with this program. It would put quite a lot of junk into an output file. A minor modification to the list function could allow listing selected portions of

John Johnson, 413 West Sycamore, Carbondale, IL 62901.

the program to an output file. This could be useful if your editor allows you to read in only complete files. You could use Browse to pull a function or two out of a larger file so you could incorporate them into a different program. I chose not to include this capability. It is not useful with my editor.

The various functions in the program are quite straightforward. I have tried to put adequate comments into the source to define each of them.

Browse recognizes several commands for moving around in the file being examined. These commands are explained with the required syntax in the function help(), which is the last function in the source listing. (See the table on page 61 for a list of the Browse commands.) Additional commands could be added by writing the appropriate functions to execute them and adding the command character into the if ... else if ... string in the function command(...). A useful command to add would be a string search capability. This could be patterned after the grep function in the book *Software Tools,* which should be in every serious programmer's library.

At this point in the development, discretion forced a halt before a simple file browser turned into a full-fledged editor. The listing accompanying this article (page 62) was created to be compiled under BDS C version 1.50a. It should convert easily to other C compilers or to other versions of BDS C.

DDJ

(Listing begins on next page)

The following is a brief listing of the commands for Browse. The information may also be found in a help screen (at the end of the program listing), which is available to the user on-line. <n> represents any positive integer.

Command	Result
<n>	Redisplay with line <n> in the center
<n> u	Go up <n> lines and redisplay
<n> d	Go down <n> lines and redisplay
	b
Redisplay beginning of file	
e	Redisplay end of file or buffer
<n> t	Reset tab stops to every <n> spaces and redisplay the screen; tab stops default to every four spaces if not set
<n> n	Go down <n> pages (default is one)
<n> p	Go up <n> pages (default is one)
q	Quit and return to operating system
<n1> <n2> l	List file from <n1> to <n2> on the system list device (line printer)

Table

File Browser Listing (Text begins on page 60)

```
/*
 *   Program Browse.c
 *   by John R. Johnson
 *   Version 1.01    Dec 7, 1983
 *
 *   Copyright 1983
 *   by John R. Johnson
 *   All rights reserved.
 *   Permission is granted for
 *   unlimited personal,
 *   non-commercial use only.
 *
 *   Address queries to:
 *
 *   John R. Johnson
 *   Professional Microware, Inc.
 *   P. O. Box 200
 *   Carbondale, Illinois, 62903
 *
 *   Reasonable telephone queries
 *   will be answered if you call
 *   at 618-529-2717.  Make sure
 *   the time is between 9 AM and
 *   5 PM Central time.
 *   I regret that I cannot return
 *   long distance calls except
 *   collect.  Thank You.
 *
 */

#include <bdscio.h>
   /* Leor Zolman's definitions */
#include <browse.h>
   /* browse definitions */

main(argc,argv)
int argc;
char **argv;
{
   int tabstop;
   /* tab stops for display */
   int lincnt;
   /* number of lines in buffer */
   int offset;
   /* window width / 2  */
   int curlin;

/*
 * active line number
 * ( center of window )
 */

int *lines;
char *max;
/*
 * array of pointers to strings.
 * This array is the line index
 * for the file.  max is the
 * maximum useable address for
 * buffer.  The buffer is
 * allocated but not accessible
 * except through this array.
 * The array lines[lincnt] is a
 * trick to get an array of
 * dynamically assigned length.
 * It is located by makbuf()
 * and created by filbuff( ).
 */

char inbuf[BUFSIZ];
/* file input buffer */
char filename[18];
/* filename buffer */
char cline[135];
/* command line buffer */

puts(CLEARS);
printf("\nBROWSE copyright ");
printf("1983 by John R.");
printf(" Johnson\n");

tabstop=4;
/* default value for tabs */

/*
 * get the file name, if it is
 * not given on the command
 * line prompt the user.
 */

if ((argc>1) &&
    (strlen(argv[1])<18))
{
   strcpy(filename, argv[1]);
}

else
{
   while(1)
   {
      puts("Enter file name > ");
      if (getline(filename, 18))
         break;
   }
}

/*
 * open the file requested
 */

if (fopen(filename,inbuf)==ERROR)
{
   error("\nfile %s not found"
                  ,filename);
   exit();
}

/*
 * create the arrays
 */

lines = makebuf(&max);
if (lines==0)
{
   error("\ncouldn't create
      buffer for %s",filename);
   fclose(inbuf);
   exit();
```

```
    }

    /*
     * read the file into the arrays
     */

    if ((lincnt=
     fillbuf(lines,max,inbuf)) == 0)
    {
       error("\ntrouble reading %s",
                          filename);
       fclose(inbuf);
       exit();
    }

    /*
     * close the file so we don't
     * do anything to it
     */

    fclose(inbuf);

    /*
     * initialize the display
     */

    offset = setoff();
    curlin = offset;

    /*
     * display the screen and parse
     * and interpret commands.
     * Notice the next current line
     * is returned from the command
     * parse routine.
     */

    while((curlin=command(filename,
          curlin,cline,offset,lines,
          lincnt,&tabstop)))
       ;

    /*
     * clear the screen
     */

    puts(CLEARS);

} /*  end of the main function  */


/*  special functions used  */

/*
 * int error(format, arg)
 * char *format, *arg;
 *
 * ring the bell and print an
 * error message formatted as
 * for the printf function.
 */

int error(format,arg)
char *format,*arg;

{
    puts(BELL);
    printf(format,arg);
}

/*
 * int setoff()
 *
 * set the display offset
 */

int setoff()
{
   return ((SCRHT-1)/2);
}

/*
 * display(keylin, offset, lines,
 *                    last, tab, f)
 * int keylin, offset, lines[];
 * int last, tab, f;
 *
 * display offset lines before and
 * after the current line.  Stop
 * when screen is full and wait
 * for further commands.  Do not
 * alter the display if current
 * line if negative.  A negative
 * value is used to keep from
 * writing over the help screen
 * when it is displayed.
 */

display(keylin,offset,lines,last,
                              tab,f)
int keylin,offset,lines[],last,tab;
char *f;
{
   int j, lone, ltwo;

   /*
    * if current line is negative,
    * omit display update
    */

   if (keylin<0)
      return;
   /*
    * don't try to display more
    * lines than there are.
    */
   if ((keylin+offset)>last)
      keylin=last-offset;

   /*
    * don't try to display before
    * the beginning of the file.
    */
   if ((keylin-offset)<0)
      keylin=offset;

   /*
    * if the entire file fits on
    * the screen, just display the
    * entire file.  Otherwise just
    * display what fits, so it the
    * current line stays in the
    * middle of the screen.
    */
   if (last<(offset+offset+1))
```

(Continued on page 66)

```
    {
        lone=0;
        ltwo=last;
    }
    else
    {
        lone=keylin-offset;
        ltwo=keylin+offset;
    }

    /*
     * do the display of the
     * lines we chose and then
     * print the file name at the
     * bottom of the screen.
     */
    puts(CLEARS);
    for (j=lone;j<(ltwo);++j)
        outlin(j,lines[j],tab);
    printf("file: %s > ",f);
}




 * int n, t;
 * char *str;
 *
 * output a line to the terminal
 * expanding tabs by t.  To take
 * care of terminals with fixed
 * tab stops.  Truncate displayed
 * line to screen width to prevent
 * uncontrolled scrolls.
 */

int outlin(n,str,t)
int n,t;
char *str;
{
    char c;
    int col,k;

    k=0;
    printf("%3d: ",n);
    /* print the line number */

    /*
     * now output the line
     */

    for (col=5;col<(SCRWID-6);)
    {
        /*
         * expand tabs for terminals
         * that don't support tab
         * setting to any widths.
         */

        if ((c=str[k++])=='\t')
        {
            do
            /* at least one blank */
                {
                    putchar(' ');
                    col++;
                } while((col%t)&&
                        (col<(SCRWID-6)));
            }

            /*
             * quit at the new line
             */
            else if ( c=='\n' )
            {
                break;
            }

            /*
             * for all others,
             * just put the char
             */

            else
            {
                putchar(c);
                col++;
            }
        }
        /*
         * always end with a new line
         */

    putchar('\n');
}


/*
 * int makebuf(pmax)
 * char **pmax;
 *
 * allocate all of free memory for
 * a buffer for the lines array
 * and the text buffer.  Arg is a
 * pointer to max so that max can
 * be set to the last available
 * memory location.  Return the
 * pointer gotten from alloc(...)
 * for the value of lines.
 */

int makebuf(pmax)
char **pmax;
{
    char *here;
    unsigned templ,temp2;

    templ=endext();
    /* first memory location */
    temp2=topofmem()-3000;

    /* last free memory */
    here=alloc(temp2-templ);
    /* allocate it */
    if (here==0)
```

```
      return (0);
   *pmax = temp2;
   return (here);
}


/*
 * int fillbuf(lines, max, inbuf)
 * int lines[];
 * char *max;
 * struct _buf *inbuf;
 *
 * Fill the string arrays allocated
 * by makbuf().  As each line is
 * placed into the text buffer,
 * enter the corresponding pointer
 * into the array lines[].  Notice
 * the text buffer builds down from
 * top of free memory while the
 * array lines[] builds up from the
 * bottom.  Filling stops on end of
 * file or when the array meets the
 * text buffer.
 * Return the number of lines read.
 */

int fillbuf(lines,max,inbuf)
int lines[];
char *max;
struct _buf *inbuf;
{
   int count;
   /* line counter */
   char *bufptr,*buflin;
   /* buffer ptrs */
   char tbuf[100];
   /* temp input buffer */

   bufptr = max;
   /* start of buffer */
   *bufptr-- = '\0';
   /* put in the null */
   *bufptr-- = '\0';
   buflin = bufptr - 8;

 if (lines > buflin)
    return (0);
 strcpy(buflin,"<Start>\n");
 lines[0] = buflin;
 bufptr = buflin;
 count=1;
 while (&lines[count] < bufptr)
 {
    /*
     * get a line into the buffer
     * and place the pointer in
     * lines[] array for access.
     */
    if (fgets(tbuf, inbuf))
    {
       buflin = bufptr -
                strlen(tbuf) - 1;
       if (&lines[count] > buflin)
          break;
       *--bufptr = '\0';
       lines[count] = buflin;
       bufptr = buflin;
       strcpy(buflin, tbuf);
       count++;
    }
    else
    {
       break;
    }

 }
 buflin = bufptr - 8;
 if (&lines[count] < buflin)
 {
    /*
     * put on the end of file
     * marker if we got there.
     */

    strcpy(buflin,"<End>\n");
    lines[count] = buflin;
    *--bufptr = '\0';
    *--bufptr = '\0';
    ++count;
 }
 return (count);
}

/*
 * int command(f, active, cmd,
 *       offset, lines, lincnt, t)
 * int active, offset, lines[];
 * int lincnt, *t;
 * char cmd, *f;
 *
 * command interpreter and display
 * handler.  Parses the command
 * line and executes the correct
 * function to execute the command.
 * Additional commands can be added
 * into the if ... else if ... else
 * construction in the parser.
 */

int command(f,active,cmd,offset,
                    lines,lincnt,t)
int active,offset,lines[];
int lincnt,*t;
char cmd[],*f;
{
   int length,first,second,temp;
   char key;

   display(active,offset,lines,
                  lincnt,*t,f);

   /*
    * If display is suppressed,
    * then activate it.
    */
   if (active<0)
      active=-(active);
```

(Continued on page 70)

File Browser Listing (Listing Continued, text begins on page 60)

```
/* get a command line */
length=getline(cmd, 132);

/*
 * if command line is null,
 * then do nothing more.
 */
if (length == 0)
   return (active);

/*
 * Set the key to the first
 * alpha in command.
 */
key=letter(cmd,length);

/*
 * Get up to two numbers from
 * the command line.
 */
first=numone(cmd,length);
second=numtwo(cmd,length);

/*
 * No key found means first is
 * the desired current line.
 */
if (key == 0)
{
   if (first<offset)
      return (offset);
   if ((first+offset)>lincnt)
      return (lincnt-offset);
   return (first);
}

/*
 * force the key to lower case.
 */
key = tolower(key);

/*
 * if key is u
 * move up n lines.
 * (default 1)
 */
if (key == 'u')
{
   if (first)
   {
      if ((active-first)
                  < offset)
         return (offset);
      else
         return (active-first);
   }
   else
   {
      if (active > offset)
         return (--active);

      else
         return (offset);
   }

/*
 * if key is d
 * move down n lines.
 * (default 1)
 */
if (key == 'd')
{
   if (first)
   {
      if ((active+first) >
          (lincnt-offset))
         return (lincnt-offset);
      else
         return (active+first);
   }
   else
   {
      if (active <
          (lincnt-offset))
         return (++active);
      else
         return (lincnt-offset);
   }
}

/*
 * if key is b
 * move to beginning of file.
 */
if (key == 'b')
   return (offset);

/*
 * if key is e
 * move to end of file.
 */
if (key == 'e')
   return (lincnt-offset);

/*
 * if key is t
 * set tabs to n (default 4)
 */
if (key == 't')
{
   if ((first<=0)||(first>20))
      first=4;
   *t=first;
   return (active);
}

/*
 * if key is p
 * n'th previous page.
 * (default 1)
 */
if (key == 'p')
{
```

```
   if (first)
   {
      if ((temp=2*offset*first)
           > active)
         return (offset);
      else
         return (active-temp);
   }
   else
   {
      if ((temp=2*offset)
           > active)
         return (offset);
      else
         return (active-temp);
   }
}

/*
 * if key is n
 * n'th next page (default 1).
 */
if (key == 'n')
{
   if (first)
   {
      temp=first*(offset+offset);
      if ((active+temp)
           > (lincnt-offset))
         return (lincnt-offset);
      else
         return (active+temp);
   }
   else
   {
      temp=offset+offset;
      if ((active+temp)
           > (lincnt-offset))
         return (lincnt-offset);
      else
         return (active+temp);
   }
}

/*
 * if key is q
 * quit and exit to system.
 */
if (key == 'q')
   return (0);

/*
 * if key is l
 * list from first to second
 * lines on printer.
 */
if (key == 'l')
{
   lister(first, second, lincnt, lines,
      *t,f);
   return (active);
}
```

```
   /*
    * default, illegal command.
    */
   /*
    * display help screen for
    * everything else.
    */
   help();
   return -(active);
}


/*
 * int letter(str,len)
 * char *str;
 * int len;
 *
 * returns the first alphabetic
 * character in string str
 * returns null if there are no
 * alphas in the string.
 */

char letter(str,len)
char *str;
int len;
{
   char c,*cptr;
   int k;

   cptr=str;
   k=0;
   while((c=cptr[k++]))
      if (isalpha(c))
         break;
   return (c);
}


/*
 * numone(str, len)
 * char *str;
 * int len;
 *
 * returns the integer value of
 * the first decimal digit string
 * encountered in string str.
 * Returns zero if no digit string
 * is found.
 */

int numone(str,len)
char *str;
int len;
{
   char c, *cptr;
   int k;

   cptr=str;
   k=0;
   while ((c=cptr[k]))
   {
```

(Continued on page 74)

```
      if (isdigit(c))
         break;
      ++k;
   }
   if (c)
      return (atoi(cptr+k));
   return (0);
}


/*
 * int numtwo(str, len)
 * char *str;
 * int len;
 *
 * returns the integer value of
 * the second decimal digit string
 * found in string str.
 * Returns zero if there is no
 * second digit string in str.
 */

int numtwo(str,len)
char *str;
int len;
{
   char c,*cptr;
   int k,n;

   cptr=str;
   k=0;
   while ((c=cptr[k]))
   {
      if (isdigit(c))
         break;
      k++;
   }
   if (c==0)
      return (0);
   cptr=nondigit(cptr+k);
   k=cptr-str;
   return (numone(cptr,len-k));
}


/*
 * char *nondigit(ptr)
 * char *ptr;
 *
 * advance ptr to the first
 * position that is not a digit
 * and return the new value of
 * the ptr.
 */

char *nondigit(ptr)
char *ptr;
{
   char c,*cptr;

   cptr=ptr;
   while(1)
   {
      c=*cptr;
      if (isdigit(c)==FALSE)
         return (cptr);
      ++cptr;
   }
}


/*
 * lister(here, tohere, topcnt,
 *                lines, tab, f)
 * int here, tohere, topcnt;
 * int lines[], tab;
 * char *f;
 *
 * Print the lines of the file
 * from here tohere on the system
 * list device using CP/M list
 * driver.  If here is greater
 * than tohere roll around to
 * line 0 when the end of file
 * is reached and continue list.
 * The listing should be paginated
 * according to the values in
 * brouse.h for form width and
 * length.  Each line should be
 * numbered.  Expand tabs by tab.
 * Print the filename and a  page
 * number at the top of each page.
 */

lister(here, tohere, topcnt,
                     lines, tab,f)
int here, tohere, topcnt;
int lines[], tab;
char *f;
{
   char c,*cptr;
int col,row,i,j,page;

if ((tohere<=0) ||
    (tohere>=topcnt))
   tohere=topcnt-1;
if ((here<=0) ||
    (here>=topcnt))
   here = 0;

i=here;
page=row=1;
col=0;

while (1)
{
   fprintf(2,"\nfilename: %s"
                           ,f);
   fprintf(2,"          page %d\n\n"
                          ,page);
   row=row+3;
   while (row<(FORML-2))
   {
      cptr=lines[i];
      col=5;
      j=0;
      fprintf(2,"\r%3d: ",i);
```

```
    while ((c=cptr[j++]))
    {
       if (c=='\t')

       {
          do
          {
             putc(' ',2);
             col++;
          } while ((col%tab)&&
             (col<(FORMW-1)));
       }
       else if ((col>=(FORMW-1))
              && (c != '\n'))
       {
          putc('\n',2);
          break;
       }
       else
       {
          putc(c,2);
          col++;

           }
        }
        row++;
        col=5;
        if (i==tohere)
        {
           col=0;
           break;
        }
        if (++i >= topcnt)
           i=0;
     }
     while (row++ <= FORML)
        putc('\n',2);
     page++;
     row=1;
     putc('\r',2);
     if (col==0)
        break;

/*
 * This comments out the pause at
 * the end of each page, reinstate
 * these two lines of code for
 * single sheet paper feeding.
 */

/*
     printf("\rpage %3d:",page);
     printf(" any key to continue > ");
     c=getchar();
 */

   }
}


/*
 * int help()
 *
 * Command help facility.  Display
 * a concise list of the browse
 * commands with syntax and
 * results shown.  If new commands
 * are added make sure you add
 * them here also.
 */

help()
{
   printf(CLEARS);
   printf("\n    Browse Command");
   printf(" Information\n");
   printf("\nCommand   Results");
   printf("\n-------   -------");
   printf("\n #        redisplay");
   printf(" with line # in center");
   printf("\n # u      go up #");
   printf(" lines and redisplay");
   printf("\n # d      go down #");
   printf(" lines and redisplay");
   printf("\n b        redisplay");
   printf(" beginning of file");
   printf("\n e        redisplay");
   printf(" end of file or buffer");
   printf("\n # t      reset tab");
   printf(" stops to every # spaces");
   printf("\n          and redis");
   printf("play screen.  Tab stops");
   printf("\n          default to");
   printf(" every 4 spaces");
   printf("\n # n      go down #");
   printf(" pages ( default is one )");
   printf("\n # p      go up # p");
   printf("ages ( default is one )");
   printf("\n q        quit and ");
   printf("return to operating system");
   printf("\n # # l    list file ");
   printf("from first # to second # on");
   printf("\n          the system ");
   printf("list device ( printer )");
   printf("\n  # represents any ");
   printf("positive integer");
   printf("\nenter any command or");
   printf(" a return to redisplay\n");
}

/*  end of the special functions  */
```

End Listing

An Introduction to Parsing

by Dr. Henry A. Seymour

What is parsing and why would a programmer want or need to learn about it? Well, parsing is the process of breaking down an input string into its most elementary parts, referred to as tokens. The portion of a program that performs this action is called the parser. Parsers are used in many areas of computing:

(1) A compiler translates a high-level language such as Basic or Fortran into object code. Usually one Fortran statement translates into about six object code instructions.

(2) An assembler translates an assembly language program into object code instructions. The assembly language is machine dependent; usually each instruction is translated into one machine instruction.

(3) An interpreter is an operation similar to the assembler, but the computer executes the machine instruction immediately; the compiler and assembler produce object code for later manipulation.

(4) A translator is a program that takes as input a source language and produces an equivalent version in the same language or in a different language; for example, Fortran 66 to Fortran 77, RPG to COBOL, etc.

(5) A pretty printer is a program that takes as input a source program, such as Pascal, and outputs the same program with indentations at the appropriate places.

(6) A command language processor is a program that accepts the job control language of the operating system and determines the meaning of the request.

(7) A query language processor is a program that accepts English language requests, determines their meaning, and performs the inquiry from a data base.

(8) A text editor is a program that accepts a string of commands and, based upon those commands, creates or modifies a file.

All of these applications are interesting enough to discuss in detail; however, because most readers are probably familiar with an assembly language, I will use the assembler as the vehicle of demonstration. Knowledge obtained in the designing and writing of a parser for an assembler should be easily applied to any of the other areas.

What smart databases, adventure games, Basic interpreters and Latin teachers have in common.

Dr. Henry A. Seymour, Martin Marietta Aerospace, P.O. Box 6184, Huntsville, AL 35806.

Assembler

In the parsing process, an input string first must be scanned to obtain the tokens of data, then the tokens must be evaluated to determine whether they are meaningful. For example, the input string

```
LOOP    LOAD    VAL,5
```

must be scanned, and the tokens must be isolated:

LOOP	LOAD	VAL	,	5

Then a decision can be made as to whether the string of tokens represents a valid instruction.

The first step in writing an assembler is to state the characteristics of the assembly language. The language that I will describe does not represent an existing language but is for demonstration purposes only. It is, however, similar to many assembly languages available today. The format is:

[label] operation operand [,register]

The brackets indicate that the enclosed field is optional. Characteristics of the language are:

- Blanks and commas are delimiters.
- The label must begin in column one.
- Each field is either numeric or alphabetic.

The second step is to analyze the language's characteristics to determine its logical structure. A graphic display of this logical structure is called a transition diagram. Figure 1 (below) shows the logical structure of the language described above. The characteristics of a transition diagram are:

- Circles are called states.
- Arrows indicate transition paths.
- Double circles mean a terminating state.

The characters associated with the arrows cause control to move from one state to another. The characters within the circles represent the state type:

- DS is the delimiter state (the beginning state).
- SS is the symbol state.
- NS is the number state.
- TS is the terminal state (ending state).

Let's go through both paths of the transition diagram to see if it will accept the language defined above. We begin at the delimiter state, DS. If the first character is alphabetic, control proceeds to the symbol state, SS, which contains the intelligence of the program. Control at SS implies that an alphabetic field is being parsed.

The looping arrow returning to SS means that control will accept any number of alphabetic characters and will remain in SS. However, upon encountering either a blank or a comma, control will proceed to the terminal state, TS. This means that the process has arrived at a point where a token has been obtained and can be saved in some location. To allow the next token to be retrieved, control is given back to the delimiter state, DS.

Now consider an alternate path: If a digit is encountered, control will proceed to NS. Being at NS implies that a digit has been found, and control will remain at NS as long as digits are encountered. Upon recognizing a blank or a comma, control will proceed to the terminal state.

The transition diagram will recognize only one unit of information at a time: a number, a symbol, or a delimiter (either the blank or the comma). This is a simplistic model that doesn't consider error conditions. I will discuss that topic later on.

The Pascal-like program in Listing One (page 82) represents the logic in the transition diagram. The functions BLANK, COMMA, ALPHA, and NUMBER test the character to determine its class. The assumption of the program is that the input record is read into an array. The procedure GET_CHAR will move a character from the input array into CHAR for later testing by BLANK, COMMA, ALPHA, and NUMBER. After the character has been tested it will be moved into the token array by the procedure MOVE_CHAR.

The point to recognize here is that the transition diagram has helped a great deal in describing the program logic. To check for more complex structures in the assembly language and for error conditions, the program must become very large and hence more difficult to read. In that case, the transition diagram will be an even more important aid in the program design process.

Although this approach to the implementation of a parser is preferable to having no developed plan at all, a better and simpler approach gives greater control over the parsing process. That approach involves one additional step: the creation of a state diagram.

State Diagram

The state diagram is equivalent to the transition diagram, but it can be implemented with greater ease and with less source code. The state diagram is a two-dimensional representation of the transition diagram. Each state—that

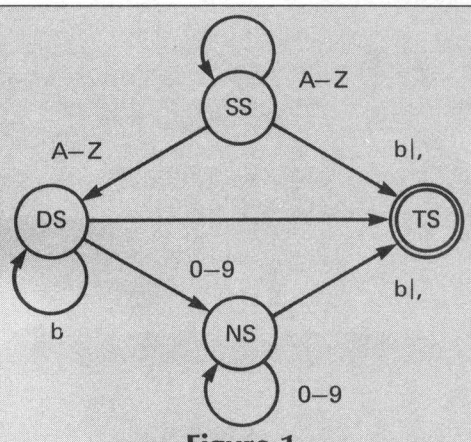


Figure 1.
Transition Diagram

State	Input Character: b	A–Z	0–9
DS	DS	SS	NS	TS
SS	TS	SS	TS	TS
NS	TS	TS	NS	TS

Figure 2.
State Diagram

is, each circle—in the transition diagram is represented as a row in the state diagram. The terminal states—that is, double circles—are the exception and are represented as rows. The transition diagram in Figure 1 would be represented by the state diagram in Figure 2 (page 79).

The contents of the array are the states that may be reached. The terminal state does not need a row because, once it is recognized, there is no reason to continue using the array. In the process of transforming the transition diagram into the state diagram, two error conditions became obvious. When in SS, no exit path exists for a digit, and when in NS, no exit path exists for a symbol. Temporarily, I will place the state code TS in the appropriate cells; later I will discuss how to specify error states.

As shown in Figure 3 (below), the input record is held in an array called an input buffer, and the parsed token is placed into a token buffer. The source record is read into the input buffer, and a pointer is used to point to each position. The program scans the input buffer, copying characters into the token buffer. Upon encountering a delimiter, the scanning process stops: the token buffer contains a unit of information.

After the token has been parsed, control is returned to the parsing process with the input pointer pointing to the character that caused the temporary halt. The process begins again at the delimiter state, and another token is parsed. The process continues until an end-of-line condition is detected.

The program in Listing Two (page 82) represents the state diagram shown in Figure 2. The function TYPE in Listing Two determines the category of the input character, as shown in Figure 2, and expresses that in the form of a column value. The program uses two variables, OLD_STATE and NEXT_STATE, to hold the state code of the token that is presently being constructed and the next state that the program is about to enter. The if statement determines how a token has been recognized. The implementation of the state diagram is efficient, easy to understand, and maintainable.

Error Conditions

The trapping of some errors can be implemented in the state diagram; however, certain errors should be checked only after the token has been obtained. For example, if the language specifications state that a label and operand variable be six characters or less, then the program must check for this. In the process of making a general but efficient model, however, some capabilities such as counting are unavailable.

The solution is to scan the input buffer, moving characters into the token buffer, until a delimiter is encountered. After the token has been obtained, its length can be determined. If the length is in error, an appropriate error message can be displayed. A typical error check for length would be:

```
FOUND : if OLD_STATE = SS and
    LENGTH(TOKEN) >6 then
        ERROR( 'LENGTH ERROR');
```

Another type of error is the combining of two tokens. In the present specifications, it is invalid to mix letters and numbers, such as LOAD5. The present state diagram will go to TS upon encountering the 5, but, upon reaching the FOUND label, the error will not be obvious. A token of 5 will be found next, and it will be up to another portion of the program to determine whether an error has been found. This kind of error causes the program to assume that the entire field has been obtained, which invalidates the remaining parsing operation. It can also cause multiple error statements to be printed when, in fact, only one error exists.

The state diagram could be rewritten to give it more error checking capability. This would be important if the programmer wanted to make the processor user friendly. How user friendly a program is depends on its ability to recognize errors, identify them to the user, and if possible make corrections. However, the more user friendly the program, the larger and more complex it is.

The state diagram in Figure 4 (at left) includes two new states, E1 and E2. The state diagram now has the ability to trap the two error conditions, such as A1 and 1A. There are no rows for these new states, and they will be treated similarly to the state TS; that is, when the program encounters E1 or E2, it will discontinue the use of the state diagram and proceed to a portion of the program that handles the error

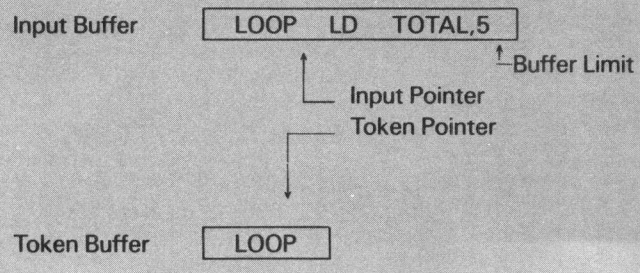


Figure 3.
Data Structures

Input Character

State	b	A–F	0–9
DS	DS	SS	NS	TS
SS	TS	SS	E1	TS
NS	TS	E2	NS	TS

Figure 4.
State Diagram with Error Traps

conditions.

The code associated with the FOUND label now can check to determine whether either of these errors has occurred and, if so, what information may be transmitted to the user. The code in Listing Three (page 86) shows one approach to this.

Expanded Assembly Language

Let's increase the strength of the pseudo-assembly language and create a state diagram that will recognize all possible valid tokens. The new instruction format is:

[label] operation operand [,register]

The characteristics of the language follow:

- Blanks and commas are delimiters.
- The operand and register may be a symbol, an integer value, or a hexadecimal value.
- The X followed by a string indicates a hexadecimal value (X'1F').
- The operation field must be a symbol.
- The label must begin in column one.
- The instruction is free form but must be stated completely on one record.
- The operand field may contain one or two operands separated by one of the following arithmetic operators: *, −, +, and /.
- The operand may contain a literal, such as RSTU.

By analyzing these characteristics, one can begin to design the transition diagram. The first noticeable characteristic is that variables are made of letters and numbers. Also, numeric, hexadecimal, and character fields all use the common alphabetic and numeric set of characters. This will be an important feature in the transition diagram.

A study of these features determines the states that must be defined, the paths between the states, and possible error conditions. The transition diagram in Figure 5 (page 83) represents a program that can recognize tokens as well as some error and warning conditions. The error and warning features include a suffix digit, which uniquely identifies the condition that has been detected. Also included is the ability to recognize the end of the input line. The meaning of each state code in Figure 5 is as follows:

- BS is the blank state (beginning state).
- DS is the delimiter state.
- XS is the X state (might be a symbol or hexadecimal state later).
- HS is the hexadecimal state.
- SS is the symbol state.
- NS is the numeric state.
- QS is the quote state.
- OS is the operator state.

In designing the transition diagram, I specified that a hexadecimal string have only the letters A through F and the numbers 0 through 9. Because the transition diagram cannot determine the running value of a hexadecimal number or a decimal number that is being parsed, this type of error trap must be expressed as source code. It is possible to trap the error condition when a hexadecimal string contains an invalid character, such as the letters G through Z. This same error can be trapped at a later point in the program. The programmer may choose where to place the trap. The error states, warning states, and terminal states do not have an equivalent row representation as do the other states. These states must be trapped by the program and appropriate action taken.

Expanded State Diagram

The columns in the expanded state diagram (Figure 6, page 84) are almost as

```
Procedure GET_TOKEN;

DS :  GET_CHAR (* from input buffer to CHAR *);
      if BLANK(CHAR) then go to DS
      else if ALPHA(CHAR) then go to SS
      else if NUMBER(CHAR) then go to NS
      else if COMMA(CHAR) then go to TS;

SS :  MOVE_CHAR (* from CHAR to token buffer *);
      GET_CHAR (* from input buffer to CHAR *);
      if ALPHA(CHAR) then go to SS
      else if BLANK(CHAR) or COMMA(CHAR) then go to TS;

NS :  MOVE_CHAR (* from CHAR to token buffer *);
      GET_CHAR (* from input buffer to CHAR *);
      if NUMBER(CHAR) then to to NS
      else if BLANK(CHAR)or COMMA(CHAR) then go to TS;

TS :  (* do work with recognized token *)
      . . .
```

Listing One

```
      OLD_STATE  := DS;
      repeat
            GET_CHAR (* from input buffer to CHAR*);
            COLUMN  := TYPE(CHAR);
            NEXT_STATE  := ARRAY(OLD_STATE, COLUMN);
            if NEXT_STATE ≠ DS then
            if TERMINAL(NEXT_STATE)  then go to FOUND
                  else
                        begin
                              MOVE_CHAR (* from CHAR to token buffer *);
                              OLD_STATE  := NEXT_STATE;
                        end
            else null (* skip blanks *);
            until INPUT_POINTER >BUFFER_LIMIT;
FOUND :           (* Token has been found.
                  Type of token is described by the contents
                  of old_state. *)
```

Listing Two

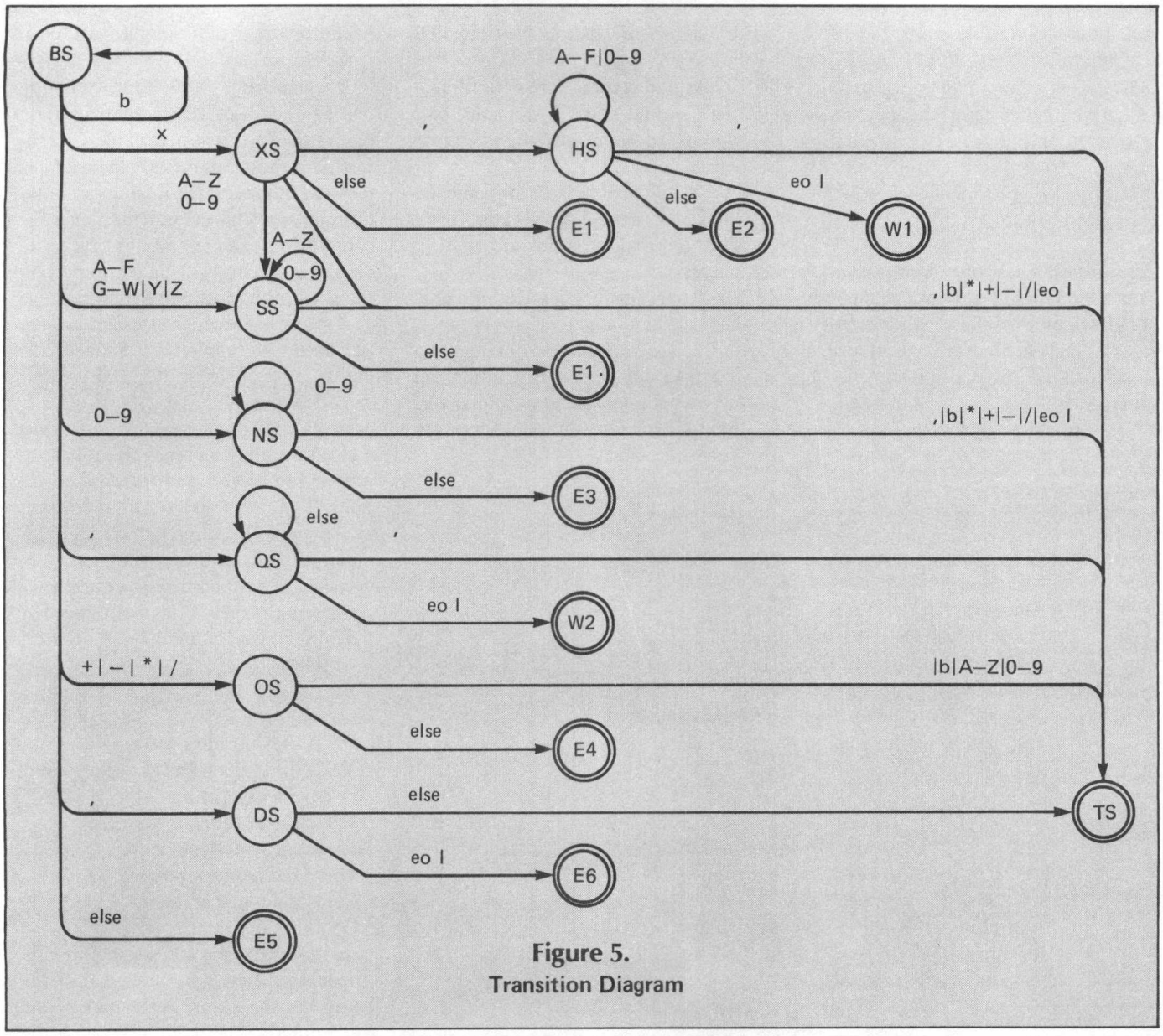


Figure 5.
Transition Diagram

straightforward as before. One must have a column for each unique type of data that is expected, plus a generic "else" column for unexpected or erroneous data. In many cases, each unique edge in the transition diagram in Figure 5 corresponds to a column in the state diagram in Figure 6.

However, as mentioned earlier, the definition of variables, hexadecimal strings, character strings, and numbers uses the same character set. The X must have a column to note the possible beginning of a hexadecimal string. Because X might also be the beginning of a variable, it will be necessary to determine whether the program has arrived at a terminal state and OLD_STATE is XS. This means that the X was recognized, the transition to state X was taken, and a symbol other than a quote caused the transition to the terminal state. In fact, the X is a symbol, and OLD_STATE must be changed to SS. A variable can include any of the alphabet and the digits, but because the digits must be separately recognized they must have their own column. The characters that make the hexadecimal string are a subset of the alphabet plus all the digits. As a result, there is no column just for variables or hexadecimal strings.

The process of combining the operators into one column requires that the program determine, at a later point, which operator it has found. It would be possible to have a column for each of the operators, but the cost of extra memory to represent them is probably not worth it.

The operand field is restricted to a simple set of cases, such as:

A
A + B
A − B
A * B
A / B

For the assembler to handle more complex arithmetic, we would have to delve into operator precedence parsing. An expression parser would build on this work, taking identified tokens as its input.

Error and Warning Messages

It is possible to have all error traps display one generic statement, i.e., IN-

VALID STATEMENT. Also the program might make no attempt to correct statements, even though it could do so, but this would not be characteristic of a user-friendly program. The error and warning messages that are referred to in Figure 6 are listed as follows:

- E1 – invalid symbol
- E2 – invalid hexadecimal character
- E3 – invalid numeric character
- E4 – invalid arithmetic syntax
- E5 – unrecognizable character
- E6 – missing register
- W1 – missing closing quote in hexadecimal (quote provided)
- W2 – missing closing quote in string field (quote provided)

When a warning state has been reached, the program should repair the string and set the variable OLD_STATE to the appropriate value. For example, if W1 is recognized and the closing quote is provided, then the OLD_STATE should be set to HS.

Master Driver

In the discussion thus far, the main point has been to obtain one token at a time. During the process, error checking determined whether a valid token had been obtained. However, it is possible to collect a string of valid tokens without constructing a valid instruction. For example,

5	X'AB'	LOAD

is a string of valid tokens, but it is not a valid instruction according to the language specifications.

We can define a transition diagram for valid tokens then convert that to a state diagram. The implementation of the token state diagram to retrieve a token will then perform as a servant to the master state diagram, which determines the validity of the input instruction. Figure 7 (page 84) shows a transition diagram for a program that determines whether instructions are correct. This master transition diagram makes use of information obtained by the token transition diagram. The transitions from one state to another use the states of the token transition diagram rather than characters. This is a more general view of the input command.

The meaning of each new state symbol in Figure 7 is as follows:

- ST is the start state.
- OP is the operation state.
- O1 is the first operand (terminal state).
- AO is the operator state.
- O2 is the second operand (terminal state).
- DL is the delimiter state.
- RG is the register state (terminal state).

In Figure 7, at only three states would it be acceptable to terminate: O1, O2, and RG. To stop while at any other state would indicate an incomplete command.

The state diagram shown in Figure 8 (page 84) reflects the logic of the transition diagram in Figure 7. I have left all the error entries blank, assuming that the reader would like to apply the knowledge gained so far by specifying the error codes. The partial program shown in Listing Four (below) is a representation of the state diagram in Figure 8.

The program begins by checking for the presence or absence of the label. If a label is present, it must be entered in a symbol table: hence the need for the special procedure, GET_LABEL. This instruction is not represented in the

Input Character

State	b	X	A–F	G–W Y | Z	0–9	'	,	+ | − * | /	eol	else
BS	BS	XS	SS	SS	NS	QS	DS	OS	TS	E5
DS	TS	TS	TS	TS	TS	TS	TS	TS	E6	TS
XS	TS	SS	SS	SS	SS	HS	TS	TS	TS	E1
HS	E2	E2	HS	E2	HS	TS	E2	E2	W1	E2
SS	TS	SS	SS	SS	SS	E1	TS	TS	TS	E1
NS	TS	E3	E3	E3	NS	E3	TS	TS	TS	E3
QS	QS	QS	QS	QS	QS	TS	QS	QS	W2	QS
OS	TS	TS	TS	TS	TS	E4	E4	E4	E4	E4

Figure 6
Expanded State Diagram

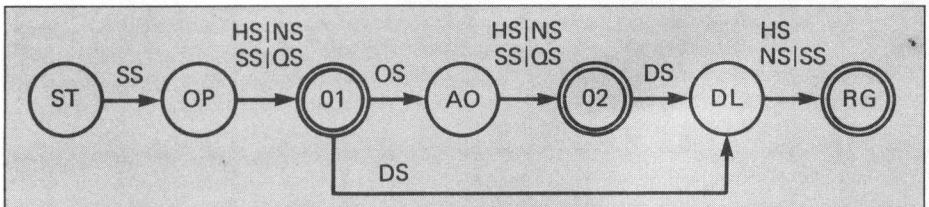


Figure 7. Master Transition Diagram

OLD_STATE

NEW_STATE	SS	NS HS	QS	OS	DS
ST	OP				
OP	01	01	01		
01				AO	DL
AO	02	02	02		
02					DL
DL	RG	RG			

Figure 8
Master State Diagram

transition diagram but is implied by the specifications of the language. The program determines at three points whether a valid command has been found. These terminating points are indicated by the double circles in Figure 7 and by the "if input_character = eol" statements in Listing Four.

Summary

The discussion about the general area of parsing has outlined an efficient method of implementing the parsing process:

- Analyze the characteristics of the input data.
- Produce a transition diagram.
- Produce a state diagram.
- Write the source code.

A method of trapping error conditions has been shown, and a coding technique that indicates uniquely which error occurred has been demonstrated. Also discussed was a method of correcting an error condition and giving an appropriate warning message. This information should be applicable to most areas of computing. DDJ

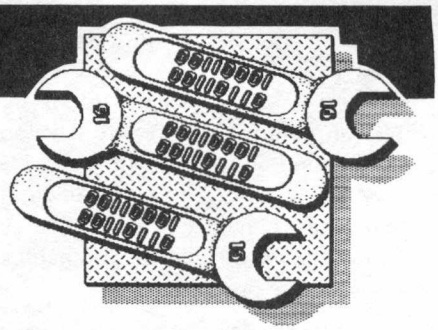
```
FOUND :     (* other work here *)
        if NEW_STATE = E1 then
        begin
            ERROR('INVALID SYMBOL');
            (* other work here *)
        end
        else if NEW_STATE = E2 then
        begin
            ERROR('INVALID NUMBER');
            (* other work here *)
        end
```

Listing Three

```
Procedure MASTER_STRING;
            . . .
            if column_one_is_not_blank then GET_LABEL;
ST :        GET_TOKEN;
            if OLD_STATE = SS then go to OP
            else ERROR;
OP :        GET_TOKEN;
            if OLD_STATE is in (HS, NS, SS, QS) then go to 01
            else ERROR;
01 :        GET_TOKEN;
            if input_character = eol then (* found acceptable string *)
            if OLD_STATE = OS then go to AO;
            if OLD_STATE = DS then go to DL
            else ERROR;
AO :        GET_TOKEN;
            if OLD_STATE is in (HS, NS, SS, QS) then go to 02
            else ERROR;
02 :        GET_TOKEN;
            if input_character = eol then (* found acceptable string *)
            if OLD_STATE = DS then go to DL
            else ERROR;
DL :        GET_TOKEN;
            if OLD_STATE is in (HS, NS, SS) then go to RG
            else ERROR;
RG :        GET_TOKEN;
            if input_character = eol then (* found acceptable string *)
            else ERROR;
```

Listing Four

16-BIT SOFTWARE TOOLBOX



by Ray Duncan

Readers Pitch M68000

The 16-Bit Mailbag brought me no less than 10 letters and cards this month from readers requesting more material on the Motorola 68000. Interestingly, not a single one of these readers contributed a 68000 programming tip, listing, or any other words of wisdom. Come on, guys, we aren't operating in a vacuum here!

Those readers who hoped that the introduction of the Macintosh would lead to the development of scads of 68000 public domain software are going to be sadly disappointed. In the first place, Apple has outsmarted itself by making program development on the Macintosh hideously difficult. The native high-level languages available for the Mac are (by IBM PC or even by Z80 CP/M standards) incredibly weak, bug-ridden, slow, and nonstandard. For example, MacBASIC can't even run the *BYTE* Sieve of Eratosthenes benchmark because the Mac runs out of memory. As its latest practical joke (or maybe this is just a nose-thumbing gesture at the free-lance software developer), Apple has released an assembler for the Mac that won't run on just one Mac—you need two. It seems a little incredible, no matter how badly the Mac's 68000 is crippled with overblown operating system software, that the self-proclaimed wizards at Apple couldn't get a two-pass assembler to run on a third-generation microprocessor equipped with 128K of RAM and a 300K+ disk drive. The old Digital Research 8080 assembler ran nicely in 32K with room left over for the operating system.

Let's travel back in time to the February 1984 *BYTE* magazine, in which Steve Jobs was quoted as saying (page 63): "This is an IBM video board; it's only video, nothing else. It's 69 integrated circuits, more chips than an entire Macintosh, and it basically does nothing. And it doesn't even do that very well." Talk about hubris! Remember, that quote was from the same guy who brought you an assembler that requires two computers.

It's becoming clear that the Macintosh's fate will be similar to the fate of the Lisa—critical acclaim, but lackluster sales. My inside source in Cupertino, Deep Golden Delicious, tells me that in the year since the Mac's announcement, approximately 200,000 machines have been delivered. Think back to all that hoopla we were bombarded with last winter, about a super Macfactory for Macs that can grind out one Mac every 15 seconds. Consider that in the same time period, more than a million each IBM PCs and Apple IIs went out the door. Now, I suppose I'm going to get piles of nasty letters from the MacWorshipper crowd. At least then we'll have 68000 topics galore to write about, won't we?

Some 8086 Debugging

Hidden in the obscurities of the Intel 8086 instruction set are some classic booby traps that can take hours (yes, even days) to debug. Here are two to watch out for.

Consider the assembly code in Listing One (page 93). This is the source for a Forth ROLL command, which picks a word out of the interior of the machine stack and moves it to the top of the stack. Clue No. 1: As written here, the command executes correctly—most of the time. Clue No. 2: In the Intel iAPX 86,88 *User's Manual* (page 2-42), the fine print says that "execution does not resume properly [after an interrupt] if a second or third prefix . . . has been specified in addition to any of the repeat prefixes." Aha! This code fails because, at unpredictable intervals, a hardware interrupt occurs during the repeated execution of the string instruction. The 8086 loses track of either the addressing context or the repeat prefix itself upon return from the interrupt (depending on the order in which prefixes were assembled); consequently, either the wrong number of words is moved, or the words are moved from the wrong source address.

Now look at Listing Two (page 93). This code is supposed to transfer a number from the top of the 8086's machine stack to the top of the 8087's machine stack. It works correctly "most of the time." The bug here results from a subtle failure of synchronization. Since the programmer failed to include a WAIT after the FLD[BX] instruction before incrementing the 8086's stack pointer, a hardware interrupt could occur and be serviced after the ADD SP,8 instruction is executed but before the 8087 has completed its transfer of the number from shared memory. Thus, the process of servicing the interrupt will push the CPU flags and return address on top of the number that the 8087 is loading, partially or completely destroying that number. This type of problem is extremely tough to isolate. Users can best avoid the problem altogether by paying scrupulous attention to synchronization and stack protection.

Sneak 80286 Preview

Since IBM has put its Good Computing Seal of Approval on the Intel 80286 with the introduction of the PC/AT, it behooves us all to start learning to use this processor properly. The 80286, when running in "Protected Virtual Address" mode, is a fearsome beast. It has several new addressing considerations, hardware-recognized data structures and descriptors, and memory protection mechanisms; it is to an 8086 what a VAX is to an LSI-11.

However, the 80286 in "Real Address" mode can be viewed as a slightly tuned up 8086; this is helpful to us aging, simple-minded software developers. Fortunately or unfortunately, PC-DOS and MSDOS 3.0 use the 80286 in Real Address mode, so we can safely ignore the more complex considerations for the present.

Changed Instructions

To start with, let's look at some subtle differences between the instruction sets of the 8086 and the 80286.

• The instruction PUSH SP pushes the current pointer, rather than the new stack pointer. In other words, the 8086 did something like this:

```
PUSH SP =
   SP    := SP - 2
   (SP)  := SP
```

while the 80286 does something like

```
PUSH SP =
   TEMP  := SP
   SP    := SP - 2
   (SP)  := TEMP
```

So the 80826 instruction

```
PUSH SP
```

has the effect of the 8086 sequence

```
MOV AX,SP
PUSH AX
```

• The 80286 divide error exception (interrupt 0) pushes CS:IP of the instruction that caused the exception. The 8086 pushed the CS:IP of the instruction following the instruction that caused the exception.

• Shift counts are masked to 5 bits. For example, if you put the value 40 in CX and execute

```
SHL AX,CX
```

the contents of AX will be shifted left 8 bits. On the 8086, the processor attempted a left shift of 40 bit positions, and the result in AX would always be zero.

Errant Instructions

There are also some known bugs in the 80286 revision B chips.

• After execution of POPF, a pending maskable interrupt may be improperly recognized, even though maskable interrupts were disabled prior to execution of POPF and the flags word popped from the stack has IF = 0. If the interrupt is improperly recognized, it will, however, be properly executed. This problem is particularly relevant for CP/M-86 system users, since many implementations of this operating system run without interrupts; if an interrupt is unexpectedly serviced, the interrupt vectors may not have been initialized, and the system will crash.

Apparently, this problem occurs only when the 80286 is running with zero or one wait states. It can be avoided altogether by running the 80286 with two or more wait states (this, however, incurs a significant performance penalty). Alternatively, you can redefine POPF in a way that simulates its action without actually executing the POPF opcode. There are a couple of similar ways to do this; here is the one given by Intel in its errata sheet:

```
CodeMacro POPF  ;assume flags on
                 stack
   PUSH CS
   CALL $ + 3    ;push IP
   PUSH BP       ;Save BP and
   MOV BP,SP     ;address the stack
                 ;add to IP value on
                 stack
                 ;to point past
                 IRET
   ADD WORD PTR [BP + 2],9
   POP BP        ;restore BP
   IRET          ;pop flags, CS, and
                 IP

EndM
```

• The LOCK prefix is ignored in instructions that reference memory only once (e.g., MOV reg,mem) but works properly for instructions that both read and write memory (e.g., ADD mem,reg or XCHG mem,reg). Thus, when programming the 80286, you should use XCHG to manipulate semaphores.

• The 80286 may fail to generate a protection exception in cases where the beginning of a multibyte operand for the 80287, addressed via DS or ES, lies within an unprotected area but crosses into a protected area. Note that this is only relevant when the system is running in Protected Virtual Address mode.

All three of these problems have reportedly been fixed in 80286 revision C parts. Revision B-2 chips can be recognized by the copyright marking of "© Intel '83."

Added Instructions

The 80286 microprocessor has several new opcodes. Most of them will be useful only to the authors of compilers or device drivers. A few of them, however, will be helpful to us average Joes, too.

Push Immediate Value

On the 80286, you can code

```
PUSH 4
```

which is equivalent to the 8086

```
MOV AX,4
PUSH AX
```

Push All (PUSHA)

This 80286 opcode will push all general registers; it is equivalent to the 8086 code:

```
PUSH AX
PUSH CX
PUSH DX
PUSH BX
MOV AX,SP
PUSH AX
PUSH BP
PUSH SI
PUSH DI
```

Pop All (POPA)

This 80286 opcode will pop all general registers from the stack; it is equivalent to the 8086 code:

```
POP DI
POP SI
POP BP
ADD SP,2
POP BX
POP DX
POP CX
POP AX
```

Note that the contents of register SP that were pushed by the PUSHA instruction are discarded rather than being loaded into SP; this, of course, is vital in saving the stack context.

Signed Multiply by Immediate Value

This was one of the more glaring deficiencies on the 8086; it is remedied on the 80286. For example, you can write

```
IMUL 10
```

where on the 8086 you would have had to code something like

```
MOV  BX,10
IMUL BX
```

Shift/Rotate Memory or Register by Count

For example, on the 80286, you can code

```
ROL AX,3
```

where on the 8086 you would have to write either

```
MOV CX,3
ROL AX,CX
```

or the sequence

```
ROL AX,1
```

```
ROL AX,1
ROL AX,1
```

Input and Output String (INS and OUTS

These are new members of the string instruction group that also includes MOVS, CMPS, SCAS, LOS, and STOS. INS transfers data from the port number in the DX register to the memory address represented in ES:DI, while OUTS transfers data from the memory address represented in DS:SI to the port number that is in DX. Both INS and OUTS can transfer either byte or word values and can accept a REPeat prefix that is controlled by the contents of CX and causes autoincrement or autodecrement of the appropriate index register, depending on the state of the direction flag.

Enter Procedure (ENTER)

This creates a stack frame and initializes a frame pointer. It was added to support the compilation of procedures in block-structured, high-level languages such as PL/I, Pascal and (God forbid) Ada.

Leave Procedure (LEAVE)

This releases a stack frame and restores the previous contents of the frame pointer. It reverses the effect of ENTER.

Detect Value out of Range (BOUND)

This tests whether an array index falls within the range defined by the contents of a two-word block of memory; if not, an interrupt 5 occurs.

There are also 16 new instructions concerned with task concurrency and memory protection that are beyond the scope of this column. They load or store global, local, or interrupt descriptor registers, control write access to regions of memory, and change task privilege levels. There is also a raft of new ways you can use familiar instructions (such as IRET) to generate protection exceptions. These we'll leave for a later, more profound, column.

New Interrupts

The 80286 adds nine new hardwired interrupts to those defined on the 8086. These are:

Interrupt Cause

5 BOUNDS executed with array index out of range

6 Execution of undefined opcode

7 Coprocessor protection error, relevant in Protected Virtual Address mode

8 Interrupt table limit fault, or the dreaded Double Fault (e.g., protection fault followed by segment not present fault, such as might be caused if the protection fault interrupt handler had been paged out by the virtual memory manager)

9 In Real Address mode, coprocessor data transfer wraparound past offset 0FFFFH. In Protected mode, this exception will also occur if the first part of a multibyte 80287 operand falls within an unprotected area but crosses into a protected memory area

10 Invalid task state segment; attempted to switch context to a task with an illegal descriptor, Protected mode only

11 Memory segment not present—support for virtual memory manager, Protected mode only

12 Stack fault—stack overflow or underflow, or stack reference to a memory segment not present, Protected mode only

13 In Real Address mode, segment wraparound attempted by a word operation at offset 0FFFFH, or a stack push with SP = 1 during PUSH, CALL, or INT. In Protected mode, general protection fault; any memory protection exception not covered by the other error interrupts.

Faster Microcode

Many of the 80286's instructions that are functionally identical to the 8086 actually execute much faster due to improved implementation. For example, a 16 × 16-bit register-register signed multiply, which requires 128 – 154 clocks on the 8086, requires 21 clocks on the 80286. This concludes our Sneak Preview of the Intel 80286 CPU. More to come in subsequent columns.

Determining PC Type

As the IBM PC family proliferates, software developers will need a way to determine the type of host machine at runtime. IBM has declared that the ROM location F000:FFFE may be inspected by software and that its content has the following meaning:

Contents	Machine
0FF	IBM PC
0FE	PC/XT
0FD	PCjr
0FC	PC/AT

It would be helpful to know what this location contains on other IBM PC-like models such as the 3270PC and the vast family of IBM-compatibles. My Compaq (an early model, ROM copyright 1982) has 02DH at this location.

If I Had a Hammer

Russ Hayden of Natick, Massachusetts, writes: " . . . in the June 1984 installment of the 16-Bit Software Toolbox, there are some 8086 assembly routines to convert binary values to ASCII hexadecimal. Ray, if these tools were screwdrivers, they would be made of tinfoil. It's not that they don't work (they'll execute fine), but rather the use of the 'DIV' (divide) instruction to divide a binary number by 16, where four right shifts will accomplish the same purpose. The DIV instruction takes 90 clocks in 8086; four right shifts take a total of eight clocks.

"I don't think we've reached the point in processing power where such things no longer matter, especially in routines likely to be incorporated into larger programs and used frequently. A column on tools should be sensitive to the many ways to approach a problem and their relative merits. [I've enclosed] a suggested improvement in the byte_to_hex routine."

It's really embarrassing to be caught out on this. I've always been a fervent advocate of using shifts and avoiding hardware divides whenever it's remotely feasible. Oh well, see Listing Three. **DDJ**

(Listing begins on next page)

16-Bit (Text begins on page 88)

Listing One

```
What's wrong with this picture?

roll proc near          ;extract word n from the
                        ;depths of the parameter
                        ;stack, pushing it on top
                        ;of the stack.
     pop  bx            ;get return address out
                        ;of the way.
     mov  ax,ss         ;can't override ES, so
     mov  es,ax         ;make it address stack segment.
     pop  di            ;get number of stack cell
                        ;to bring to the top.
     mov  cx,di         ;calculate number of stack
     inc  cx            ;words to slide.
     sal  di,1          ;calc destination address
     add  di,sp         ;for slide.
     mov  si,di         ;calculate source address
     sub  si,2          ;for slide.
     push ss:[di]       ;copy the desired cell to
                        ;top of stack.
     std                ;set direction flag for
                        ;string move.
                        ;now slide the stack.
     rep movs es:word ptr [di],ss:word ptr [si]
     add sp,2           ;clean up stack pointer.
     jmp  bx            ;return to caller.
roll endp
```

End Listing One

Listing Two

```
What's wrong with this picture?

.
.
.
mov  bx,sp
wait
fld  ss:[bx]
add  sp,8
.
.
.
```

End Listing Two

Listing Three

```
Improved conversion routine (see also column in June DDJ).

byte_to_hex proc near          ;convert binary value to
                               ;hex ASCII
                               ;AL=binary value
                               ;DI=pointer to storage
                               ;   for string
     mov  ah,al                ;save lower nibble
     shr  al,1                 ;divide upper nibble by 16
     shr  al,1
     shr  al,1
     shr  al,1
     call ascii                ;convert it to ASCII
     stosb                     ;and store it
     mov  al,ah                ;get back lower nibble
     and  al,0fh               ;mask to four bits
     call ascii                ;convert and store
     stosb
     ret

byte_to_hex endp
```

End Listings

(Continued from page 22)

which host buffer contains the requested sector and where the sector is within the buffer. If the request is a read, the host buffer is read into memory and the logical sector extracted. The buffer is kept intact in hopes that other sequential read requests will be made that can be satisfied directly from memory.

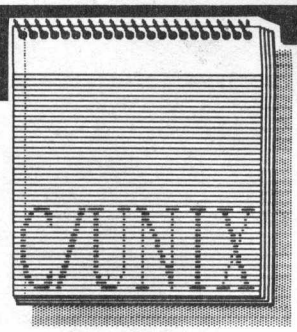
On the other hand, write requests are a little more difficult. Not only must the system place the record to be written into the proper slot in the host buffer, but any other data in the same buffer must not be disturbed. This usually requires that the host buffer first be read into memory, then modified by inserting the logical sector, and finally written back to disk. As before, the action of restoring the updated buffer on disk is held up in hopes that another sequential write request will be made—which, again, would be handled at memory-to-memory speed.

Everything that I have talked about works very well most of the time. A few situations, however, create severe buffer conflicts requiring two or three times more disk I/O activity than would be required on a single-density system. The cause of this problem is the bottleneck created by using only one memory-resident I/O buffer. For example, many traditional batch processing programs are written to input an old master file and write an updated one based on maintenance transactions input from a third file; each time a record is written to the output file, the output host buffer must first be read from disk, updated with the new logical record, then written back to disk. The additional I/O activity is due to the necessity of repeatedly writing the host buffer just updated back to disk to make room for the next incoming master file and transaction file record. This amounts to two extra I/O operations that wouldn't be required on a single-density system.

Several methods have been used to alleviate this problem. One allocates an extra memory buffer reserved exclusively for write operations. CP/M Plus also uses buffer memory, almost as much as you can give it, to buffer host sectors. Even at its worst, using larger physical sectors will give you a much more responsive disk system.

DDJ

C/UNIX PROGRAMMER'S NOTEBOOK



by Anthony Skjellum

In this column, we'll consider some reader feedback on material presented in earlier columns. First, I will mention some new volumes available from the C User's Group. Following this, I'll present some corrections to errors in previous columns.

CUG Volumes

Machine-readable software can help you avoid a lot of frustration. To help readers of this column, I have created two C User's Group volumes: CUG *DDJ*, Volumes 1 and 2. They contain material up to and including the October 1984 column. The group's new address is:

C User's Group
415 East Euclid
McPhearson, KS 67460
(316) 241-5450

Volumes are available in several formats, including IBM PC and popular 5¼-inch CP/M-80 formats (e.g., Osborne DD). Contact Robert Ward at the above address for details.

Runge-Kutta Correction

Three errors were evident in the October column. The differential equation used in the example (from page 94) should have been

$y'(t) = 1 + t - y$

and not

$y'(t) = 1 + y$

as printed. Furthermore, the solution to this equation is

$y(t) = t + 5.0*\exp(-t)$

for the initial condition $y0 = 5.0$.

Also, the display( ) function on page 98 of the October issue is corrected in Listing Three (page 102) of this issue.

X Grammar Examples

Some of the examples presented in the September column were set incorrectly. The errors principally involve the omission of semicolons at the ends of certain lines. For example, in Figure 1, page 116, there should be a semicolon following the word COMPLEX. Furthermore, both assignment statements in the first cadd( ) function of Figure 2 are missing their semicolons. Figure 3 is missing its terminating brace. Finally, an errant semicolon appears on the first line of Figure 5.

Long Pointer Corrections

I presented a Long Pointer package in the June column. Bruce Komusin wrote from Monaco to point out some errors in the assembly language routines. He writes:

"I just read your article in *DDJ* #92 about long pointers for C. I never [have] used C, but I know 8086 assembler. From your listing of llsup.asm, it is apparent that you overlooked the fact that the 8086 affects the flags when doing INC or DEC [instructions] for 16-bit [quantities]. This is a com-

```
ldec            proc near
                or          bx,bx
                jnz         ldec_1
                mov         ax,es
                sub         ax,1000h
                mov         es,ax
ldec_1:         dec         bx
                ret
ldec            endp
```

Figure 1

```
vec int sort_them(argcnt, argvec)
int argcnt;
int *argvec[]; /* Integer arguments */
{
        /*
         * exchange the values of the argvec array
         * here
         *
         */
}
```

Figure 2

```
vec int sort_them(argcnt, argvec)
int argcnt;
int *(argvec[0])( );
int *argvec[];        /* The rest are integers */
{
        /* etc. */
}
```

Figure 3

mon error because it is different on the 8080. So, for example, you can save bytes and time in the routine linc by removing the OR BX,BX."

While this concerns only inefficient coding, Mr. Komusin continues to point out a real bug:

"However, I really wrote this letter to warn you about ldec. Of course, it will not work as is because of the DEC BX changing the zero flag set up by the OR BX,BX. I suggest a change . . . that fixes everything."

The change is presented in Figure 1 (page 96). Beyond the basic fixes, Mr. Komusin suggests some increases in efficiency:

"However, here are some points about execution speed and byte efficiency. It is much faster to 'fall through' a conditional jump than to actually jump. So, if possible, it is always a good idea to arrange the code so that the normal case falls through and only the exceptional case jumps. As a side benefit, the exceptional case can then be shared."

A full set of improved routines are presented in Listing 1 (page 97). I want to thank Mr. Komusin for his letter and corrections.

Programming Philosophy

John A. Grosberg of Scottsdale, Arizona, wrote an interesting letter concerning programming style and philosophy. He wrote his letter after reading the August 1984 column, which included a short listing by Alex Cameron. Mr. Grosberg writes:

"Your column . . . caught my attention, particularly the short listing of Mr. Alex Cameron's routines to automatically allocate I/O buffers (page 119). I am writing to present a few ideas on program structure and will use his listing as an example. This is not an attack on his application or on the style he used in his listing—I assume that there were reasons for the form chosen. But my perfectionism was provoked by that listing, and the more I read it, the more I wanted to write.

"One important principle of program design is that the structure of the program (I will use *program*, *routine*, and *function* interchangeably for this discussion) should reflect the structure of the problem. This sounds nice, but what does it mean? Without guidelines it is almost a theological principle, over which well-meaning people could argue loud and long and never come to agreement. The reason for this is that the 'structure of the problem' depends on one's viewpoint; i.e., it is relative to the observer . . . the program's structure reflects the way we are thinking about the problem."

Since the way we write programs is based on our viewpoint, Mr. Grosberg suggests a set of standard reference points:

"In mechanical drafting, there are three standard orthogonal viewpoints that are used to describe most objects. They are called 'front,' 'side,' and 'top' views of the object. The structure of the physical object inheres in the spatial relationships of its elements, and these must be captured in the drawing.

"In software, an important aspect of structure is the temporal relationships among the elements. The two primary temporal relationships are sequence and frequency, and these relationships should be captured in the code. If one action occurs before another in time (sequence), then the first should precede the second in the code. If an action

```
vec char *dunno(argcnt, argvec)
int             argcnt;
char            *argvec[0];
long            *argvec[1];
double          *argvec[2];
COMPLEX         *argvec[3];
int             *argvec[];      /* rest are integers */
{
        /*
         * Here we have a function whose first four arguments
         * are respectively: char, long, double, and COMPLEX
         * pointers. Anything after that is an integer
         */
}
```

Figure 4

```
;
; improvements to llsup routines by Bruce Komusin
; % Microworld
; L'Estoril
; 31 Ave. Princesse Grace
; Monte Carlo, Monaco
;
; these routines offer more temporal and byte-efficient
; code than those originally presented in llsup.asm
;
ldec            proc            near
                or              bx,bx
                jz              ldec_2
                dec             bx
                ret
ldec_2          dec             bx
ldec_3          mov             ax,es
                sub             ax,1000h
                mov             es,ax
                ret
ldec            endp
lsub            proc            near
                sub             bx,ax
                jb              ldec_3
                ret
lsub            endp
```

Listing One

occurs the same number of times (frequency) as another, then they should be in the same (logical) block of code."

I think that Mr. Grosberg's recommendations are practical. In my opinion, this type of coding technique could only improve maintainability of software. He continues:

"Expanding on the concept of temporal relationships as expressed in code, consider that on any single execution of a program, an element of that program may be executed once, more than once, or less than once [i.e., not executed]. If the element executes once and only once per program execution (sequence), it should simply be listed in sequence where it belongs. If the element executes more than once per execution (repetition), it should appear once in a loop. If the element executes less than once per program execution (alternation), it should appear once in a program branch statement. Finally, all elements that execute the same number of times should appear together in the listing."

While these points seem obvious to me (and also to Mr. Grosberg), it is clear that they are not often followed. I cannot claim to have adhered to these principles in the past, although I plan to do so in the future. For those interested in pursuing the concepts further, he recommends *Practical LCP, a Direct Approach to Structured Programming*, by Albert C. Gardner (McGraw Hill, 1981). Mr. Grosberg has recoded Alex Cameron's listing to exemplify his comments; this code is presented in Listing Two (page 98). His comments concerning the code itself follow:

". . . in Mr. Cameron's function 'sfopen,' the call to 'alloc' actually occurs only once per execution, but it is written three times in the code [sequence]. The 'return' occurs only once per execution, but is written 10 times [sequence]. The three main 'if' statements

```
if(*mode == 'x')
{
    ...
}
```

are written as if they occur sequentially, when in fact only one of them can occur per execution [alternation]. The structure of the code actually obscures the execution behavior of the function."

Mr. Grosberg doesn't claim to have embodied his comments perfectly in Listing Two. He just created it (untested) to illustrate his remarks. I found the ideas worthwhile.

Another Response to the August Column

Mike Meyer writes the following concerning the August issue:

"I just read the August '84 column. Applause—you struck a solid, well-balanced position. If you can continue doing as well, you'll have a great column.

"Let me ask a favor: Please, PLEASE avoid the 'Unix/C is better/worse than <*fill in blank*> because <another blank>'-type arguments. As you said, your column 'exists for discussing C and Unix as they are, with the problems they have.' Such discussions don't fit into that mold and tend to generate more heat than light."

"Couple of nits: James Jones and Jeff Bowles are at uokvax, not ea. Their net address should be ucbvax!mtxinu!ea!uokvax!emjej, jab}. Finally, my last name is Meyer, not Meyers. If you could cease pluralizing me in the future, I'd appreciate it."

I want to apologize for this error. Mr. Meyer has been a regular corre-

```
#define NULL        0
#define BUFSIZ      256
#define TRUE        1
#define ERR         -1

sfopen(filename, mode)
char *filename;
char *mode;
{
        int fd,
            err;
        if (fd = alloc(BUFSIZ))
        {
                switch(*mode)
                {
                        case 'w':           /* write mode */
                                err = (fcreat(filename,fd) == ERR);
                                break;
                        case 'r':           /* read mode */
                                err = (fopen(filename,fd) == ERR);
                                break;
                        case 'a':           /* append mode */
                                err = (fappend(filename,fd) == ERR);
                                break;
                        default:                    /* invalid mode */
                                err = TRUE;
                                break;
                }
                if (err)
                {
                        free(fd);
                        fd = NULL;
                }
        }
        else                                /* alloc failure */
        {
                fd = NULL;                  /* redundant */
        }
        return(fd);
}
```

Listing Two

```
/*
Revised <untested> version of "sfopen." The original version was written by A. Cameron
and published in DDJ, August 1984.
This revision is to illustrate the design principle that the structure of a program should reflect
the behavior of the program (a corollary of the principle that the structure of a solution should
reflect the structure of the problem).
The focus of this example is that the execution sequence and execution frequency are primary
elements of problem structure and should be mirrored in the code.
by John A. Grosberg
[relevant for BDS C]
*/
```

spondent, which means I had ample opportunity to see his name and reproduce it properly. (I'll get his name straight from now on.) He continues:

"To add some constructive comment, I'd like to point out that relying on library utilities for things does not guarantee portability. For instance, many C implementations won't have the Unix math(3) library or the qsort(3) routines. Of note is that the current AT&T Unix distribution doesn't include the dbm(3) routines from Unix version 7. I use those routines to fix the 'everything is line-oriented ASCII' problem with Unix, and some of the AT&T sites that don't have that library complained when they got copies of my software."

This is an interesting point that I had not considered. It adds more complexity to the idea of C/Unix software portability. What libraries can and cannot be assumed when writing a program? Is it OK to think of libraries such as CURSES as standard?

Comments on the X Grammar

I received several comments about the X grammar. In this column, I present one letter; the rest are reserved for the February 1985 column [*DDJ* No. 100]. John M. Gamble of Batavia, Ohio, writes:

"Your column on extensions to the C language was very interesting. I have a few comments.

"(1) To keep analogy between functions and opers, I think that it should be legal to declare static opers."

This sounds fine, but what is a static oper? Since I'm not sure what Mr. Gamble means, I can't really comment. He continues:

"(2) I have trouble thinking of any justification for adding one more reserved word (loop) just to do what 'for(;;)' does just as well. If it really offends your eye, couldn't you just use #define to substitute for it?"

I agree. I only mentioned "loop" because I wanted an efficient way to specify an unconditional loop. This is fine since "for(;;)" shouldn't produce unnecessary instructions in object code. Mr. Gamble continues his list of comments as follows:

"(3) I think that your method of declaring argument lists in vec functions is too limited to be practical. A function list is not analogous to argv, which deals only with character strings. A function, after all, deals with all sorts of variables. To get around this problem, I have thought of two possible solutions:

"(a) Require that the first argument be a string equivalent to printf's control string. Quite frankly, I dislike this solution. Deciphering the control string would be a pain, and the code needed to deal with this pain would probably ruin C's reputation for compact code.

"(b) Declare the types of the argument list members in the function itself. This would be efficient and easy to modify later on. For example, say that you wish to have some integer variables, and you wish to exchange their values so that they are in [numerical] order. Rather than going through the trouble of inserting the values in an array, calling a sorting routine, and recovering the values from the array, a vec function called sort_them( ) might be easier to use. The declaration might be as depicted in Figure 2 (page 96).

"If you wanted to make the function more flexible by allowing the ordering to be user-specified, we could have the first argument be a function. Then the declaration would resemble Figure 3 (page 96).

"Of course, we are not limited to integer pointers. A vec function could just as easily have arguments of all sorts. Such an example is presented in Figure 4 (page 97).

"Since the argvec array consists of pointers only, the addresses of the argument list are passed in, not the values. Therefore, register variables may not be used in the list. Also, since passing addresses is the default, we can drop the '&' before each variable [in the calling sequence]."

I think Mr. Gamble's ideas are valid and are consistent with my original intentions for an extended grammar. Does anyone else have further comments about vec functions?

Conclusions

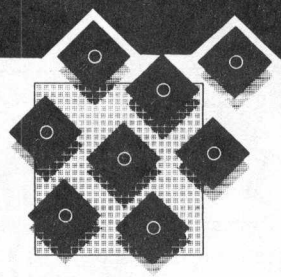
I have wound up the year by including some corrections and comments about previous columns. The programming structure comments offered by Mr. Grosberg were particularly interesting to me, and I hope that others find them useful as well. I was pleased with Mr. Gamble's suggestions concerning C extensions, and I look forward to additional remarks in this area.

My next column will appear in the February 1985 *DDJ*. This is issue No. 100, a landmark for the journal. In this column, I'll include additional reader feedback and suggestions as well as possible topics for future columns. Have a happy holiday season.

DDJ

```
/* display( ): subroutine to print an ascii file on the console */
display(fname)
char *fname;
{
        char c;         /* character to output */
        FILE *disp;
        if((disp = fopen(fname,"r")) == NULL)
                return(-1);   /* can't open file */
        while((c = getc(disp)) != EOF) /* print the file */
        {
#ifndef UNIX
                if(c == TEOF) /* text end of file */
                        break;
#endif
                putchar((c & 127));   /* output each character less parity */
        }
        fclose(disp); /* close the file */
        return(0);      /* successful completion */
}
```

Listing Three



NCI COHERENT

Company: Network Consulting Inc., Suite 110, 3700 Gilmore Way, Burnaby, B.C. Canada V5G4M1
Computer: IBM PC and IBM XT
Price: $695

Reviewed by A. Gomez

COHERENT was developed by the Mark Williams Co. as a Unix look-alike that did not have to pay royalty fees to AT&T. NCI has modified COHERENT with a number of real-time enhancements to enable efficient multiuser operation with small machines. The kernel is ROMable and may be embedded in peripheral modules and software for communications packages. The utilities for the most part were rewritten in assembly language for speed of execution. Finally, specific hardware is supported that allows expedient processing in a Unix style environment.

Features

The features of NCI COHERENT fall into two parts: hardware and software support. Hardware support is the support of a specific item of hardware that allows flexibility of configuration or enhances the performance of the system. Software features are those that provide ease of programming or system use. By this definition, a hardware feature may be a piece of software (e.g., a device driver).

The hardware features of NCI COHERENT are:

- Choice of hard disk support: NCI COHERENT provides device drivers for the XT, CORVUS, CORONA, DAVONG, TECMAR, GENIE, EAGLE, INTERPHASE SMD, MAYNARD SASI interface, and COLUMBIA disk controllers and drives. Note that this feature allows NCI COHERENT to operate on compatibles as well as the PC itself.
- Ziatech IEEE-488 interface: The 488 interface is important in control applications.
- Interactive Data Systems 1600 BPI tape drive interface: This device driver allows the use of 9-track tape with NCI COHERENT.
- Tall Tree Systems JRAM card: This card is frequently used as a RAM disk because of its ability to window itself in an already RAM-filled PC. It has 512K of RAM on each card and can increase performance of swapping systems by being part of the file system.
- Persyst 4-line serial port card with onboard 8088: This allows the main processor to be off-loaded and permits the suitable execution of real-time applications in a multiuser environment.
- Control Systems Hostess 8-line serial port card: This card is necessary to provide the 11-user capacity that is claimed by NCI. The card supports XON/XOFF and variable size words (5 to 8 bits) in either interrupt or polled mode.
- Hayes autodialer (or any other autodialer with programming effort): This provides support for the "cu" program.
- Support of the AST real-time clock via the "clock" command.
- Support of three parallel line printers.

The software features of NCI COHERENT are:

- Kernel parameters: NCI COHERENT provides the ability to change the kernel parameters via commands (e.g., partitioning of RAM disk).
- TERMCAP: This package will allow programs that are screen based to be terminal independent. It is widely used in the Unix community.
- 165 Unix V7 commands: These commands are rewritten in assembly language for speed.
- Lex and Yacc: A lexical analyzer and compiler compiler, commonly found and used on Unix systems, are of great value to developers who need to create parsers and lexical analyzers.
- RCS: This is a source code control system.
- Unix SYSTEM V memory routines: These routines are supported in the kernel and include memccpy,

Benchmark	Compile Time (sec)	Execution Time (sec) Real	User	System
Pipes	30	27.4	0.1	22.0
System call	19	29.5	2.8	26.4
Function call	18	0.5	0.3	0.1
Sieve	19.6	8.5	8.1	0.2
Disk write	32.1	7.8	0.1	6.3
Disk read	33.0	16.0	0.1	7.0
Shell	–	13.0	1.4	5.8
Loop	18.2	26.6	26.4	0.1

Multiprocess benchmark	# Processes (1)	(2)	(3)	(4)	(5)	(6)
Multi.sh	15.1	21.3	29.1	36.6	45.8	52.1

Table 1

memchr, memcmp, memcpy, and memset.

- Prolog interpreter: The prolog interpreter is frequently used in artificial intelligence work and control applications.
- An IEEE floating point math package that is accessible by the Unix math library routines.
- A screen editor ("see"): This is not the VI or EMACS, with which all of us are familiar, but a screen editor.
- License fees: The single-user price is the multiuser price; no additional fees are necessary. Furthermore, runtime only versions are available for applications with embedded kernels and offboard processors. This allows software developers to use the facilities of the NCI COHERENT kernel in their product without having the customer pay the full price of NCI COHERENT.
- Coming soon is a MSDOS bridge that allows a MSDOS program to run under NCI COHERENT.

Documentation consists of a reference manual (sections 1 – 8 of the Unix manual), a tutorial manual (ed, m4, lex, rcs, etc.), and the book *Using the UNIX System* by Richard Gauthier. The documentation was complete and organized in a manner consistent with other Unix systems.

Unix Compatibility

NCI COHERENT was designed to be compatible with Unix V7, and it looks like NCI has achieved its goals. All the system calls are identical, the libraries are identical, and only minor functions or user programs are different.

To check its compatibility, I moved various applications between a V7 Unix, the NCI COHERENT, and a SYSTEM V Unix. Without exception, all worked without error. Of course, these programs did not use the additional features of SYSTEM V, but then I was checking for V7 compatibility.

Benchmarks

Nothing is more controversial than benchmarks in product marketing. It is really hard to choose a set of programs that presents a fair comparison of different products. The benchmarks that I used in measuring NCI COHERENT were those found in the July 1984 issue of *Byte* magazine in an article named "Benchmarking UNIX Systems" by David F. Hinnant. These benchmarks appear to be fair and, more importantly, to provide a set of measurements upon which to grade the performance of NCI COHERENT. In that article, Mr. Hinnant provides timings for nine benchmarks in 15 different systems. Table 1 (on page 106) shows the measurements of NCI COHERENT for the same benchmarks.

By comparing NCI COHERENT to other Unix systems on the PC, we see no improvement in the single-user environment. However, as the number of active processes increases, the degradation of NCI COHERENT is less than other Unix systems.

Closing Notes

In general, I was impressed by NCI COHERENT. NCI has placed its emphasis on the performance of a system with more processes than that found in a single-user environment. Although I did not test the multiuser capability of the product, I do not doubt that it exists because of the orientation of the kernel. The price is reasonable, considering other offerings for the IBM PC, and the company looks like it's heading toward full compatibility with Unix SYSTEM V, which allows a future for its products and support.

Turbo Pascal Version 2.0

Company: Borland International, 4113 Scotts Valley Drive, Scotts Valley, CA 95066, 1-800-227-2400 ext.968 outside California, 1-800-772-2666 ext.968 in California

Computer: MSDOS, PCDOS

Price: $49.95

$89.95 with 8087 support

Reviewed by Karl R. Kachigan

Well, it just had to happen. Friends were talking about it, the computer shows featured it, and just about every magazine beamed about its significant performance for the price. Just what kind of decent Pascal could this be for $49.95? Originally, because it didn't support graphics on my IBM PC, I ignored it, continuing to use my UCSD p-System Pascal with its Turtlegraphics and other goodies. Then version 2.0 was announced—it supported graphics, color, sound, windows, overlays, and was still only $49.95. I nibbled. I telephoned in my order, and about a week later I became a believer. For the price and its features, Turbo Pascal compares quite well with my UCSD p-System Pascal.

Before you wonder how I was converted, let me say that Turbo Pascal is truly a good solution for both the novice and the experienced programmer. When Turbo Pascal is properly installed, and that isn't a difficult task, you have a nicely integrated programming system. The editor/compiler lets you compose a program, compile it, and either run it or return to the editor to correct errors flagged by the system, all without using the disk.

To those unfamiliar with the UCSD p-System, its editor, filer, and compiler are integrated but usually not simultaneously resident in memory (unless you configure a RAM disk that can simulate this). The UCSD Pascal compiler/editor obviously was the model for Turbo Pascal. Both indicate syntax errors with descriptive messages and will point to them if you return to the editor. Turbo Pascal just seems faster and slicker at it; plus, for those of us using the IBM PC, it runs under PCDOS. Only recently has the UCSD p-System been capable of doing this. However, the UCSD p-System does provide more capability, machine control, and transportability—but at a much higher cost.

This review should give you a better feel for the version 2.0 enhancements, especially on the IBM PC. I will build upon David Clark's review of Version 1.0 in the June 1984 issue of *DDJ*. Please reread this article if some of my points seem unclear. Because many of the fancy additions (the manual calls them "IBM PC goodies") are specific to the IBM PC and its compatibles, I will try to point out what features are applicable to all machines and which are IBM specific. Table 2 (page 110) lists all of the enhancements in version 2.0.

What Is Turbo Pascal?

Turbo Pascal is distributed as either one or two disks (the second is for MSDOS/PCDOS 8087 support), two manuals, several programs including sample programs, and an update file giving last minute information. Turbo

Pascal itself is composed of either two or three files: compiler/editor (TURBO.COM, TURBO.CMD, TURBO-87.COM, or TURBO-87.CMD), error message data file (TURBO.MSG), and for CP/M-80 only an overlay file (TURBO.OVR). A terminal installation program, provided to customize the Turbo Pascal screen control and editor features, includes the program itself (TINST.COM or TINST.CMD), a message data file (TINST.MSG), and a terminal data file (TINST.DTA, not on the IBM PC version). Provided on all versions is a program lister (TLIST-.COM or TLIST.CMD) and a sample spreadsheet program called MicroCalc (supplied in source code form). Included with the MSDOS/PCDOS versions are information files on using MSDOS function calls, external assembly language routines, and interrupts, each with comments and a sample program listing. The IBM PC version adds sample programs on color, graphics, sound, and windows—all in source code format.

The manuals, an updated version 1.0 manual and a short version 2.0 supplement, attempt to cover all versions of Turbo Pascal (CP/M-80, MSDOS, PCDOS, and CP/M-86). They are typeset, seemingly well structured, and indexed.

Standard Turbo Pascal uses real numbers with a range of 1.0E–38 to 1.0E+38 and 11 significant digits. With the 8087 support, these expand to 4.19E–307 to 1.67E+308 and 16 significant digits. As for RAM requirements, version 2.0 for MSDOS takes 35K and for PCDOS 36K, compared with 33K for the old version 1.0 16-bit implementations.

Editor Enhancements

I've mentioned that the built-in editor is a nice feature. A full screen editor, it requires specific information on your terminal's control sequences and editing commands/keystrokes. Here is where the terminal installation program is helpful. It has two modes: select (or define) the terminal used and define the edit command keystrokes. For the non-IBM versions, a large selection of the most popular terminals are listed. If you are using one on the list, just select it. If yours is not listed, you can add your terminal to the list by entering the appropriate information, such as the cursor addressing, screen clear, insert/delete line, and enhance on/off control sequences. In either case, Turbo Pascal is now installed for the selected terminal.

I installed the MSDOS version on my HP-150 by adding my terminal to the list and found the installation program quite accommodating. It was gratifying to see a program that could adapt to my HP-150's lengthy cursor addressing sequence (typically nine characters with ASCII row and column addressing). The IBM PC version lets you select from six display modes: default monitor, monochrome monitor, mono or color 40 columns, and mono or color 80 columns.

As for the editor, 45 commands are listed, almost all of them standard WordStar features. The version 2.0 manual claims to have added seven new commands, but they were also listed in the version 1.0 manual. (Apparently these were documented before they were actually implemented.) I felt right at home using the familiar WordStar control sequences again. This is truly a nice touch, especially having the same block move/copy/delete and find/search/replace capabilities. Not only does the IBM PC version use the WordStar commands, but version 2.0 also hooks up all the IBM PC keys, such as insert/delete, page up/down, cursor pad, home/end, and many more.

For the 8087 version, please note that you must temporarily rename the TURBO-87 file to TURBO if you want the TINST program to install your terminal selection with the right runtime program. After running TINST, you can change the updated TURBO file back to "TURBO-87" to distinguish it from the standard TURBO.

Speed

Okay, here's where Turbo Pascal shines, compared with the other Pascals on the market. I wondered how Borland could be so brash in their advertisements claiming Turbo Pascal compiled in seconds while others compiled in minutes. Quite simply, you edit a file by bringing it into RAM then selecting the memory compilation mode, after which Turbo Pascal compiles from RAM just as if you were using a RAM disk—at blazing speed with no disk accesses.

I had configured my UCSD p-System on my IBM PC to do this, and it sure makes Pascal a fun language to use: I don't even mind the iterations of edit, compile, find the missing semicolon, re-edit, and so on. Turbo Pascal makes this RAM disk-like feature transparent to the user by defaulting to it. Of course, you can also compile to a disk file for execution outside the Turbo Pascal environment (i.e. standalone applications or programs).

A simple test of compilation speed was to compile one of the sample IBM programs, ART.PAS, a graphics demo. With my stopwatch in hand, I determined that the 151-line program compiled in 3 seconds! Longer programs such as the MicroCalc demo also compiled in an amazingly short time. The painful part of Pascal—compilation and syntax correction—has been removed. Even novices will appreciate that their learning time can be spent more on the language than on compilation syntax checking.

As for execution speed I unfortunately couldn't locate an 8087 to check Turbo Pascal's true speed, but I can compare common benchmarks running on my 4.77 MHz IBM PC and my 8 MHz HP-150. Using the *DDJ* Savage floating-point benchmark that first appeared in November 1983, I produced some new entries for Ray Duncan's floating-point benchmark table that appeared in *DDJ*'s August 1984 issue. This benchmark really tests the round-off errors and significant digits of a language, plus its math speed. Like Turbo Pascal version 1.0, version 2.0 still doesn't include an arctangent function, so I had to do it in math.

As you can see in Table 3 (below), Turbo Pascal version 2.0 hasn't really increased its speed on the IBM PC from version 1.0; the HP-150 excells primarily due to its 8 MHz CPU. While Turbo Pascal doesn't show blazing execution speed, it is very accurate in its math.

The Reference Manuals

My only complaint with the manual is that, in trying to cover the CP/M-80, PCDOS/MSDOS, and CP/M-86 variations in one manual, the summary ta-

bles occasionally were incomplete for the 16-bit versions: the tables represented the CP/M-80 implementation correctly but failed to really indicate the additional procedures and functions of the 16-bit implementations. The manual is divided into sections: general information, CP/M-80 specific, MSDOS/PCDOS specific, CP/M-86 specific, and summary tables. In the 16-bit sections covering the MSDOS/PCDOS and CP/M-86 function calls, no syntax definitions were given at all. I really had to study the documentation files on the distribution disk to get the necessary information. This may have been due to an initial uncertainty in their implementation, but this is the second edition of the version 1.0 manual. I hope that Borland will update this manual soon to clean up its errors.

As for the version 2.0 manual, it is short and simple. The descriptions of the overlays, graphics, color, sound, and 8087 enhancements are adequate. Once again, the summary table could be more helpful in indicating which version 2.0 additions are IBM PC specific, plus be more accurate. Both of the window procedures and the draw procedure were improperly listed. No mention is made of any differences between the CP/M-80 and 16-bit implementations of overlays and the dispose function, so I trust both work similarly.

Compiler Directives

Nothing new here except that the X directive (array index optimization) is CP/M-80 specific and not used in the 16-bit versions. Also, the I directive (I/O error checking on/off) uses the standard Pascal function IoResult, not IoError. Both of these are corrections to David Clark's version 1.0 review.

New Enchancements

Here's a simple overview of what version 2.0 adds. For all versions:
(1) Overlay system (with code swapping to write programs larger than the memory available for program code)
(2) Dynamic heap (with a real dispose function to supplement the more restrictive mark and release procedures of version 1.0)
(3) Additional editor commands
For IBM PC and compatibles only:
(4) Colors
(5) Graphics
(6) Windows (graphics and text)
(7) Sound
For 16-bit versions:
(8) Optional 8087 support

Overlays

How many times have you composed a program that ended up being too large to compile? Many large application

Enhancement	IBM PC Only
Reserved word	
overlay	
procedure	
Dispose(var P:Pointer);	
FreeMem(var P:Pointer, I:Integer);	
TextMode(Color:Integer);	XX
TextBackground(Color:Integer);	XX
TextColor(Color:Integer);	XX
GraphColorMode;	XX
GraphMode;	XX
HiRes;	XX
GraphBackground(Color:Integer);	XX
Palette(Color:Integer);	XX
HiResColor(Color:Integer);	XX
Plot(X,Y,Color:Integer);	XX
Draw(X1,Y1,X2,Y2,Color:Integer);	XX
Window(X1,Y1,X2,Y2:Integer);	XX
GraphWindow(X1,Y1,X2,Y2:Integer);	XX
Sound(I:Integer);	XX
NoSound;	XX
function	
MaxAvail:Integer;	
WhereX:Integer;	XX
WhereY:Integer;	XX
pre-defined constants	
4 for text mode selection	XX
16 for color selection	XX
1 for blinking text	XX

Table 2
Turbo Pascal Version 2.0 Enhancements

Computer	MHz	Language	Vers	FPP	Time (sec)	Error
IBM PC(8088)	4.77	Turbo Pascal	2.0	8087	??	??
IBM PC(8088)*	4.77	Turbo Pascal	2.0	--	535	4.6E-3
HP-150(8088)	8	Turbo Pascal	2.0	--	396	4.6E-3
IBM PC(8088)†	4.77	Turbo Pascal	1.0	--	544	5E-3

* Same results with PCDOS and MSDOS versions of Turbo Pascal
† As tested by Jeff Furgal

Table 3
Turbo Pascal Benchmarks
for June 1984 DDJ 16-Bit Toolbox

programs are like this. Programmers usually resort to breaking the program into smaller chunks that are independent in function to the others. If you think this sounds like the definition of a procedure or function, you're right, except that it is on a larger scale: groups of procedures and functions are the chunks. Some familiar terms and techniques to achieve this are overlays, chaining, separate compilation, segments, units, external modules, and so on. Let's take a quick look at the most popular techniques to see how Turbo Pascal and its overlays fit in.

Chaining is best exemplified by Microsoft BASIC, whether it is on CP/M, MSDOS, or the IBM PC. Fundamentally, every program has a runtime library of built-in functions plus variable/data space. One program can "chain" to another (i.e., call the other program and relinquish control) with little difference in the programs beyond executing a chain statement. Chained programs also typically like to pass information to one another without using a disk file—via common variables. Common variables mean that the intepreter or compiler sets aside a specific amount of variable space, protecting it from being destroyed in the chaining process. If the chained programs have equivalent common-variable definition statements, these variables aren't garbage or initialized but retain their values for use by the new program. Key advantages are in passing data; also, with compiled programs, the runtime library need not be loaded again.

Separate compilation and modules let you write parts of the program separately, each with different variables, types, procedures, and functions. After all the separate modules are compiled, a main program (sometimes called a shell program) calls each module as it needs it. This may entail loading all the modules in memory at once or doing memory management and swapping in and out of memory only the modules that are most recently used. Both MS-Pascal and UCSD p-System Pascal implement variations of this. In fact, the UCSD p-System itself is an excellent example of separate compilation; many of its users quickly notice the disk swapping of the numerous parts of its system in and out of memory as needed. Key advantages are that you can write and compile many chunks of a program independently of the others to compose a rather large application.

Overlays are similar to both chaining and separate compilation. As implemented in Turbo Pascal, you may compose a program with as many overlay groups in the source code as you want and compile it all at one time. An overlay group is a sequential list of procedures and functions that can be called independently of the others in the list. Each overlay group sets aside an amount of memory equal to the largest procedure or function in the group, then swaps into this space the procedure or function you call. To use one or more overlay groups in the program, you effectively define which routines don't call each other, then group them together in their own overlay. Key advantages are code swapping for minimal program memory.

In Turbo Pascal, you define an overlay group with a sequential list of routines using "overlay" preceding the function or procedure declaration. Note that an overlay group ends when a procedure or function definition without the overlay designation is encountered. Hence, you can set up several overlay groups, separating each group with a non-overlay routine—even a dummy procedure of begin/end will do it.

Listing One (page 117) shows a program composed of two overlay groups, each with two procedures. Note that a disk access is performed with each call to a new procedure in an overlay group. As Turbo Pascal compiles your program, each overlay group is compiled to disk under the name of your program with a file type of .000, .001, .002, and so on. This means 1000 overlay groups are possible for a single program! The number of routines per overlay group seems to be unlimited. To actually compile a program with overlays, you must select compilation to the disk; Turbo Pascal can't handle it in its own memory. Likewise, to run the compiled program, you must exit Turbo Pascal and execute the program from DOS—definitely an inconvenience.

The advantages of overlays in Turbo Pascal are in saving program and data space, especially since overlays can be nested. The disadvantages are that you can't compile to memory, the program and overlay files must reside on disk, you can't execute a program while in Turbo Pascal, extra time is needed for the disk I/O for overlay retrieval, extra code is generated for overlay management, and the runtime debugger has problems inside the overlay groups.

I have a greater appreciation of the UCSD p- System separate compilation capability (units and segments) after using Turbo Pascal's overlays. In essence, Turbo Pascal's overlays arè the equivalent of UCSD p-System's segments. I have grown fond of the UCSD p-System unit concept, especially for generating a large application program. For smaller jobs, however, the segment/overlay approach is more than adequate.

Dynamic Heap Management

Many lanaguages have the capability to dynamically allocate and deallocate variables and, more importantly, memory space. For example, Microsoft BASIC has its string variable garbage collection. Pascal has two approaches: mark/release and new/dispose. Turbo Pascal version 1.0 implemented the mark, release, and new procedures, but no dispose procedure. Version 2.0 now fully implements the dispose procedure, a nice enhancement.

Why the fuss about dispose? Let's look at how different the two Pascal approaches are, then you'll see why. Imagine some sequential stretch of memory where variables are stored. Under the mark/release approach, you specify a "mark" variable as the memory pointer. When you later execute the release procedure with this "mark" variable, it and all variables stored after it are erased. Hence, mark/release defines where new variable space starts by deleting a block of previous varibles. New/dispose works a bit differently: these procedures use and make available variable space in a random fashion. When you "dispose" a variable, the memory space used by that variable can be reused when a new variable is specified by the "new" procedure.

In summary, mark/release specifies a block of RAM for reallocation, and new/dispose specifies any variable

space for reuse. One last comment: mark/release and new/dispose should never be used together—you must select the approach you wish to use.

IBM PC Goodies

Now let's get to the meat of the version 2.0 enhancements: monitor control, graphics, color, sound, and windows for the IBM PC and its compatibles (see Table 2). Borland has been nice to us here; I just hope they can support other computers with these types of enhancements.

Monitor control includes four modes:

(1) Text mode—25 lines of 40 or 80 characters, color or black and white
(2) Graph color mode—320 x 200 dots, four color graphics
(3) Graph mode—320 x 200 dots, black and white graphics
(4) HiRes mode—640 x 200 dots, black and one color graphics.

In text mode, after selecting the appropriate display format, you can define both the background and the text colors using the color graphics board. You can select from any of 16 available colors. Whichever monitor you use, the "where cursor" functions give you the current x and y coordinates of the cursor. You can even have the text blink by selecting the text color with a blink offset (add 16).

In the graph color mode, you have control over the background and pen colors. After selecting the background from one of 16 available colors, you define the pen color according to which palette you are using (there are four). For example, palette(0) is composed of background, green, red, and brown. This will be the most used graphics mode.

Graph mode is similar to graph color mode, except that background color is black and pen color is white. If you are using an RGB monitor, this mode is enhanced: you can select a background color from the 16 available and a pen color from one of two palettes.

In HiRes mode, you select only the pen color; the background is always black, independently of whether your monitor is color or black and white. The pen color is selectable from the 16 available.

Now, how do you use graphics? The plot procedure basically plots a point on the screen. The draw procedure plots a line between two points. In both cases, you have control of the pen color. The x and y coordinates for each point are defined with the origin (0,0) in the upper lefthand corner of your monitor; the lower righthand corner becomes either (319,199) or (639,199), depending upon the resolution used.

This orientation is the same as IBM BASIC, but different from my UCSD p-System Turtlegraphics where resolution control is the same, but the lower lefthand corner is (0,0) and the upper-righthand corner is (319,199) or (639,199). I feel more comfortable with the latter, but most of us can use either approach.

The area fills, circles, etc., of IBM BASIC are missing here, but these are not a great loss. One feature that I do miss from my Turtlegraphics is the ability to rescale the screen to use any coordinate system I wish. With a single statement, I could set the x-axis of a graph to be the years 1900 to 2000 and the y-axis to be dollars from $1,000 to $10,000. To do this in Turbo Pascal, you have to define a special procedure to scale every x and y coordinate pair before you plot or draw. Also remember that, because the x and y coordinates must be integers, you must truncate whatever your scaled values are back to integers or Turbo Pascal will not accept them.

The first thing I tried after plotting the trace was to label my graph. You do this by overlaying text and graphics—with a few limitations. Characters can be placed only in the grid of 24 lines by 40 characters using the GotoXY procedure, and the text color will be whatever the current pen color is. My Turtlegraphics lets me put a character or string at any point in the graphics screen. Graph labeling could be nicer, but at least Turbo Pascal has the same capability as IBM BASIC.

Plotting speed is comparable to both IBM BASIC and UCSD p-System Turtlegraphics—not slow. Listing Two (page 119) illustrates a simple graphics program: a sine wave with axes and a title for the plot. It took about 34 seconds to plot everything, most of which was math computation time. When I removed the x and y computations from the for/next loop and just

Turbo Pascal, Version 2.0

Company: Borland International, 4807 Scotts Valley Drive, Scotts Valley, CA 95066
Computer: CP/M-80, or CP/M-86, CCP/M-86
Price: $49.95
Reviewed by David D. Clark

Before my original review of Turbo Pascal, version 1.0, made it into print, Borland announced a new release. Because the original was so good, I was eager to get my hands on the new one. Again I have not been disappointed. The revised product has several improvements, most of which will have the biggest impact on users of the IBM PC and compatibles. That version is reviewed by Kachigan starting page 107, but the editors of *Dr. Dobb's* let me look over the new 8-bit version as well.

The editor has been expanded slightly with some additional commands, still similar to WordStar's. The standard procedure, Dispose, has been implemented. An additional new function, MaxAvail, is provided that returns the amount of memory available for allocation of dynamic storage. The FreeMem procedure is now provided to release blocks of dynamically allocated memory from the heap. It is symmetrical to the GetMem procedure previously provided.

The big news for the 8-bit version is the inclusion of dynamic overlay facilities. Truly huge programs can be built with this facility. It allows you to specify procedures or functions that can be read into memory when required. It works this way: In the source file, the new reserved word "overlay" is placed before the procedure or function keywords of those subroutines that you wish to run as overlays. Groups of subroutines declared consecutively in the program source will be placed in the same overlay file after compilation. At runtime, routines from the same file will occupy the same space in memory and will be retrieved from disk as they are invoked. To place overlays in different overlay files, the routines must be separated by the declaration of a nonoverlay object. A dummy type declaration will do. Nested overlays are allowed to any depth.

This overlay strategy places some restrictions on their use. Since overlays

in the same file run at the same location in memory, they may not call one another, although overlays placed in a separate file, running at a different memory location, may. Also, since memory space for overlays is allocated statically during compilation, the space in memory that they would occupy is not available for anything else; for example, the heap does not increase in size when no overlays are executing. Finally, programs with overlays must be compiled to disk. You will cause compiler error if you try to compile such a program in memory.

The space requirement for this new flexibility is quite modest; it now requires 28K of memory to run Turbo on an 8-bit system instead of 27K. The 16-bit version, with windowing and all that other stuff, now takes 33K. In either case, it is still an incredibly small system.

The first time I reviewed this product, the only thing I could find to complain about was the documentation. It wasn't bad, just a little rough. It's improving, it's improving. Apparently some of the errors in the programming examples were corrected before version 2 was announced. Some other revisions have been made to the manual as well. Some sections still grate on my ears, and some words are still broken at odd places, but the basically good manual has gotten better.

Because of the improvements to the system, I discovered a shortcoming that I hadn't noticed in the original release: there is no Exit procedure. Let me explain. Because of the new overlay capability, I transferred a big, nasty, variable step size, variable order, variable method numerical integration program written in UCSD Pascal to CP/M for use with Turbo. That program was originally written in fairly unstructured Fortran. When the program runs, it is often necessary to return from a deeply nested group of procedure calls without "unwinding" back through them all. Standard Pascal can do this with a goto statement to any location in a program. UCSD Pascal allows gotos only within a block, just like Turbo Pascal. As an alternative, UCSD provides a standard procedure called Exit, which will exit from the subroutine named as its single argument. It is used when unwinding from a deeply recursive set of calls would be clumsy. Turbo does not supply such a mechanism.

The global goto of standard Pascal is difficult to implement on a one-pass compiler such as the Turbo and UCSD products. An Exit procedure is fairly simple, though. It just involves traversing the chain of activation records until it passes the one sought. If anyone at Borland is listening, an Exit procedure would sure be nice.

In summary, I found version 2.0 of Turbo Pascal to be incrementally better than its predecessor. The documentation is being polished up. The editor has been expanded slightly. A couple of useful new procedures have been included. The overlay facilities add a lot of flexibility to an already excellent product. Maybe most important, these enhancements have not made the system large and unwieldy. Turbo Pascal is still small, fast, a pleasure to use, and only $49.95.

plotted a fixed point, the program took less than 2 seconds!

Some final comments: Selecting any of the graphics modes will clear the screen; if you wish to clear the graphics screen, you must reselect the current graphics mode (the ClrScr procedure works only in text mode). If you have both monochrome and color monitors, you must work on only one of them—you cannot control both simultaneously. And, finally, there is no way to get a hard copy of the current graphics screen. If I could add a few things, I'd like to be able to rescale the x and y axes and to have a graphics dump to my printer. These would make Turbo Pascal a superb graphics tool.

Windows! We do windows! Yes, Turbo Pascal has windows for text and windows for graphics. In graphics, GraphWindow defines what part of the screen you can plot on. If you draw a line that crosses through the window, only the pixels within the window are affected; everything outside it remains unchanged. In text mode, Window defines where the upper left screen coordinates (1,1) are. All console output appears only within the window, long lines wrap around the window, and the GotoXY function even works.

Sound or NoSound now beep, play songs or make a siren (this is actually part of the SOUND demo program). Quite simply, you specify the tone frequency in Hertz with the Sound procedure, vary the time between tones with the Delay procedure, and turn off the sound with the NoSound procedure.

Conclusions

I am well pleased with Turbo Pascal. It sure comes close in many areas to my UCSD p-System Pascal. However, many of the standard Pascal features missing in version 1.0 are still missing here. They are:

(1) The Get and Put procedures for I/O and the file buffer variables do not exist. You must use the extended capabilities of Read, Readln, Write, and Writeln to handle all I/O. When translating from other Pascals, this will be a large stumbling block.

(2) The Goto statement is restricted to transferring control only within the current block, not outside of procedures or functions.

(3) The Page procedure is not implemented.

(4) Variable packing and unpacking are not within your control; Turbo Pascal decides itself. Hence, the reserved word Packed has no effect, and the procedures Pack and Unpack are not implemented.

(5) Procedures and functions cannot be passed as parameters to other subroutines.

Although every Pascal attempts to follow Jensen & Wirth as a guideline to standard Pascal, very few can restrain themselves from deviating on some functions and adding embellishments. Turbo Pascal adds its share of niceties to Pascal: The UpCase function and screen commands (ClrScr, ClrEol, etc.) are a real help. Also, a simple access to CP/M and MSDOS/PCDOS function calls is provided via a procedure and register record.

I'd recommend Turbo Pascal to anyone interested in Pascal, whether novice or expert. When I talked with one of Borland's technical wizards, he clued me in to a version 3.0 that is coming in the next six months. A major planned addition is complete separate compilation capability. If that happens, I'll probably put away my UCSD p-System Pascal for good. It still shines in better screen control (via screen_ops), communications support

(rem_unit), more enhanced graphics (Turtlegraphics), and separate compilation (units), but I'm sure the folks at Borland can come close.

While I'm discussing what's missing, let me give other UCSD p-System Pascal people a quick review of what needs to be translated from your UCSD p-System programs to make them Turbo Pascal programs. Not implemented:
(1) units
(2) exit
(3) pwroften(*)
Modifiable:
(1) io_result—different error number
(2) segments—overlays
(3) screen_ops—various screen functions
(4) close—different syntax
(5) reset—must use assign and reset, different syntax
(6) rewrite—must use assign and rewrite, different syntax
(7) turtlegraphics—quite a bit restricted
(8)strings—don't default to 80 characters long, there is no default length at all
(9) underscore—Turbo Pascal doesn't ignore them, they become significant characters in names.

DDJ

Software Reviews (Text begins on page 106)

Listing One

```
program over;
  {this demonstrates Turbo Pascal's overlay features.
     File:   over.pas, over.com

    there are two overlay groups in this program
     Files:  over.000 --- proc1a, proc1b
             over.001 --- proc2a, proc2b   }

type
  string80 = string[80];

var
  s:  string80;

overlay procedure proc1a;
  begin
    writeln('This is the first procedure in overlay #1');
    readln(s);
  end;

overlay procedure proc1b;
  begin
    writeln('This is the second procedure in overlay #1');
    readln(s);
  end;

procedure dummy;
  {this forces the two separate overlay groups}
  begin
  end;

overlay procedure proc2a;
  begin
    writeln('This is the first procedure in overlay #2');
```

```
    readln(s);
  end;

overlay procedure proc2b;
  begin
    writeln('This is the second procedure in overlay #2');
    readln(s);
  end;

begin
  clrscr;
  writeln('Disk access for overlay group #1');
  proc1a;
  writeln('Disk access for overlay group #2');
  proc2a;
  writeln('Disk access for overlay group #1');
  proc1b;
  writeln('Disk access for overlay group #2');
  proc2b;
  writeln('Done');
end.
```

End Listing One

Listing Two

```
program graph;
  {demonstrates Turbo Pascal graphics}

  var
    s: string[80];
    pi180: real;
    i,x,y: integer;

  begin
    graphcolormode;       {set 320 x 200 color graphics}
    graphbackground(0);   {set background to black}
    palette(0);           {colors are black/green/red/brown}

    draw(0,0,0,199,1);
    draw(0,199,319,199,1);
    draw(319,199,319,0,1);
    draw(319,0,0,0,1);      {draw a frame}

    draw(0,100,319,100,1); {draw center line}
    gotoxy(6,2);
    writeln('This is a Turbo Pascal Graph');

    pi180:=3.14/180;
    for i:=0 to 720 do
      begin
        x:=trunc(i/2);
        y:=trunc(100+75*sin(i*pi180));
        plot(x,y,2);        {draw a sine wave}
      end;

  end.
```

End Listings

(Continued from page 17)

the Shugart. The Shugart door handle opens left; the Tandon opens to the right. The Shugart will seek to, and write on, track 77 (the 78th track), but the Tandon will go not-ready if you try that. I know, because my disk format routine had an off-by-one error that didn't show up until I tried it on the Tandon.

"Anyway, that's how I lost a drive and some downtime and about $600. I have my half-heights now, though, even if they are of different makes. Plus an extra drive I can cannibalize for spare parts—poor headless thing."

The Intern tossed the crumpled coffee cup toward a box of bloody listings. "Thanks for listening." He wandered off down the hall, a rumpled figure under the cold lights. The wind howled outside, and the rain streamed down the windows.

DDJ

COMPUTER CALISTHENICS

by Michael Wiesenberg

Bob Levin, hotshot young programmer at I-Q Industries in the heart of Silicon Valley, stands at one of I-Q's permanent coffee stations and stares forlornly at the empty coffee pot. The burner has been on all night and all that remains in the pot is a dark brown paste.

Grey Scrivener, senior technical writer, glides around the corner. He smiles as much at Levin's predicament as at his tattered tennis shoes, wrinkled wash pants, and TEX t-shirt ("!You can't do that in horizontal mode"). Scrivener smooths an invisible wrinkle in his Calvins and takes over. He carries the pot into the men's room and washes out the muck. When he returns, Levin has been joined by Spotswood Gilbert, senior programmer, who holds a raisin Danish in one hand and a chocolate brownie in the other.

Marian Smith, system operator, arrives, trailing a cloud of smoke, which Scrivener fans away with his hands. She sets her cigarette on the edge of the table. Burn marks on the table indicate that she has probably done this before. "Baby Huey's down. You been messing with the operating system again, Bobby?"

"The changes I make only speed the system up. If it's down again, you can blame all the unsupported utilities. You'd think a company as big as this would get a full-screen editor for program development."

Sally McRae walks up, having overheard the last part of the conversation. She has on Nikes, baggy shorts over purple tights, and a Cocolat t-shirt. "Maybe you didn't crash the system, but you sure brought it to its knees. What're you doing to use up all that cpu time?"

Levin, uncomfortable around attractive women, is too embarrassed to look McRae in the eyes, so he settles on her nose. "I'm trying to solve a puzzle."

Gilbert takes the last bite of his Danish and starts on the brownie. "Is it as tough as that last one?"

"Nah. This was a contest by a puzzle magazine. Winner got a new IQ2557Q portable computer."

Smith notices the veneer top of the coffee station table starting to char and shoves the cigarette back in her mouth. "Hey, I'd like one of those. Even with the employee discount, they're not cheap."

"Well, the funny thing is I don't think the winner used a computer to come up with the winning answer. I think a computer could have found a better solution."

Scrivener adjusts his Stanford ring. "You mean this puzzle had more than one answer?"

"No, but I'm pretty sure there's a better answer than the one that won the contest. I just don't think any of the entrants found it because I don't think anybody knew how to write a program to solve the problem. I wish I'd heard about it before the contest was over. I have a program running right now that should give me the answer that would have won. Listen:

"Assign a numerical value to each letter of the alphabet, starting with 1 for A and going up to 26 for Z. Any word in the English language has a value obtained by multiplying the values for each of its letters. For example, the word *hello* is worth 86,400, obtained by multiplying 8 × 5 × 12 × 12 × 15. Which English word is equal to exactly 1,000,000? If there is none, which is closest? Only words found in *The Random House Dictionary of the English Language* (unabridged edition) can be used. No capitalized words, none with hyphens or other embedded punctuation, nor those designated as foreign."

McRae laughs. "How about a program that does my work for me at noon while I drive to Gelato Classico?"

Gilbert finishes the brownie and pours the last of the coffee. "Hmm. I can see right away that the word has to have at least five letters."

Scrivener dumps the old filter and grounds, measures fresh coffee into a new filter, replaces the holder, and presses the BREW button. "I see that. The four-letter combination with the highest value would be *zzzz*, and that multiplies out to considerably less than a million."

Gilbert presses buttons on his watch. "How about *linger*? That's 952,560."

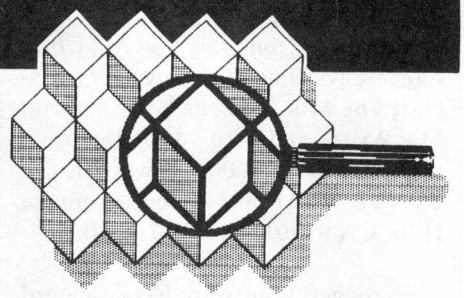
Levin pushes the straggly blond hair off his forehead. "That's 47,440 off, and you're just guessing. I can give you a better guess: *single*. That's 1,005,480, only 5,480 off. But there's a scientific way to figure it, and I'm sure it can be done only by computer."

* * * * *

Well, how about it, folks? Can you devise a program that finds the right word? And, having done that, can you tell us what that word is? Your program must be short and elegant. The algorithms can be demonstrated in a good pseudolanguage if you wish or perhaps in flowcharts. The best solution wins a t-shirt and will be published here.

DDJ

OF INTEREST



by R. P. Sutherland

Unix Directories

The Unix market is growing at a frantic rate. The number of licenses for Unix and Unix imitations is a quarter of a million and that figure is projected to quintuple over the next two years! Two Unix software directories containing 400 sources each can put folks in touch with suppliers of Unix applications software. Onager Publishing has announced the second edition of ***Unix Applications Software Directory***. Cost: $50.00. Onager Publishing is at 6451 Standridge Ct., San Jose, CA 95123 (408) 225-3541. Another resource is the ***Unix System Encyclopedia***. In addition to alphabetical and categorical listings of sources and programs, a third of the book includes articles that describe hardware systems and industry trends. ***The Unix System Encyclopedia*** is priced at $34.95, available from Yates Ventures, 4962 El Camino Real, Suite 111, Los Altos, CA 94022 (415) 964-0130.

Unix Utilities

A $90 **Modula-2** system for the Fortune 68000 system running Unix is available from Modula-2 Corporation. The system allows 128K of workspace for Modula-2 programs, which may be divided between code and data in any ratio. The compiler and interpreter are supplied with the basic I/O modules described by Wirth in *Programming in Modula-2*. Modula Corporation is at 1673 West 820 North, Provo, UT 84601 (800) LILITH2.

A **Writer's Workbench** for Unix System V and 4.1 and 4.2 Unix systems is available from International Data Services. Writer's Workbench consists of 25 computer programs that will do such things as proofread text, analyze spelling and punctuation, and check for sentence length, structure, and voice. Writer's Workbench provides on-line information about English usage and allows users to establish their own standards for text analysis. Writer's Workbench is distributed at a cost of $2,000.00. Call International Data Services at (408) 986-1972.

C Tools

Complete Software, Inc., and Catalytix Corporation, both of Cambridge, Massachusetts, have announced some interesting C tools. Complete Software has introduced a **menu-driven C language debugger** that operates independently of host systems. CDEBUG is an interactive tool that symbolically debugs C source code routines on any software or hardware that runs C. Priced from $300.00, CDEBUG is available from Complete Software, Inc., 60 Aberdeen Ave., Cambridge, MA 02138 (617) 492-5305.

Catalytix Corporation has a checkout compiler for the C programming language as well as a C interpreter. The **Safe C Compiler** adds runtime checking to C programs. The **Safe C Interpreter** provides interactive execution of C programs. Catalytix Corporation is at 55 Wheeler St., Cambridge, MA 02138 (617) 497-2160.

C Compilers for Macintosh and Lisa are available from Softworks Limited and Consulair Corp. Softworks Limited has a triple pass compiler system. The C library includes system interface functions, Unix functions, and complete interface to all Macintosh ROM routines. The Lisa implementation will produce programs for either Macintosh or Lisa. The package is $395.00 for Macintosh and $695.00 for Lisa. Contact Softworks Limited, 607 W. Wellington, Chicago, IL 60657 (312) 975-4030.

Consulair Corp. is shipping a C compiler and support library for the Macintosh (or Lisa under MacWorks). Use of Mac C and the Mac C Toolkit requires Apple's Macintosh 68000 Development System (assembler/debugger). Mac C is $295.00 and Mac C Toolkit is $175.00. Consulair Corp. is located at 140 Campo Drive, Portola Valley, CA 94025 (415) 851-3849.

Apple Stuff

A cross assembler that allows development of MC68000 assembler programs on Apple II computers has been made available for $100.00. Allen Systems' **SX-68 cross assembler** is written in 6502 assembler. Complete access to DOS 3.3 as well as the instruction set specified by Motorola for the MC68000 are supported. Allen Systems, 2151 Fairfax Road, Columbus, OH 43221 (614) 488-7122.

DOS 4.0 for the Apple II family has been released by Rune Software for $95.00. The company claims that DOS 4.0 offers better performance than DOS 3.3 because it employs a new CMOS 6502 instead of the Apple II's existing NMOS 6502. Advantages include faster processing, increased disk storage capacity, as well as eight new processor instructions. For a full list of features, contact Rune Software at 80 Eureka Square, Suite 214, Pacifica, CA 94044 (415) 355-4851.

Assembly language programming on the Mac for the Mac is possible with **MacASM**, a co-resident editor/macro assembler. MacASM integrates the editor, assembler, linker, resource

compiler, and debuggers so that they can be used from the MacASM full-screen environment. Source files generated by MacASM can be edited with MacWrite, and vice versa. Introductory price is $100.00, from Mainstay, 28611B Canwood Street, Agoura Hills, CA 91301 (818) 991-6540.

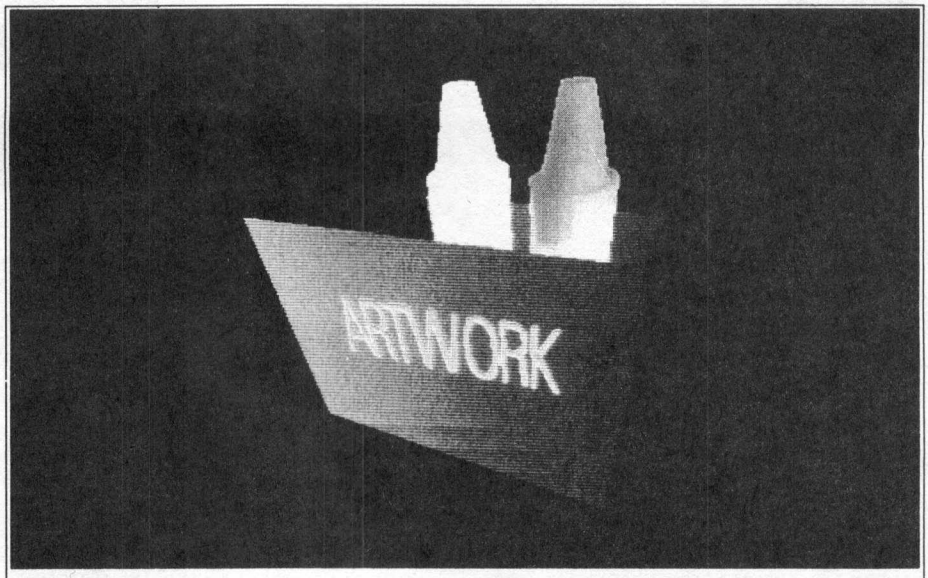
Pterodactyl Software has developed **PC BASIC**, a Basic compiler for the Apple Lisa. PC BASIC is syntax compatible with BASICA on the IBM PC. This product allows Basic programs to be quickly converted to run on the Lisa. In addition, PC BASIC provides nearly unlimited core memory space for programs and data, and allows developers to link programs to Lisa's graphics operating system, as well as Lisa's Pascal and 68000 programs. Cross-compiled and native mode versions for the Macintosh will be available soon. The price for one protected copy of the compiler is $250.00, or $750.00 for a runtime license that allows one to include the runtime package with applications for resale. Contact Ed Rosensweig at (415) 485-0714.

Miscellany

DriveLiner

Chandler Software has developed a portable CP/M program for verifying alignment of 8-inch floppy disk drives. A Dysan Diagnostic Diskette (see Loren Amelang's article in the December 1983 *DDJ*) is supplied with the program. Head centering, radial, and azimuth alignment tests are performed automatically on any CP/M 2.2 compatible 8-inch floppy system. The price is $65.00 from Chandler Software, 273 West Shore Drive, Marblehead, MA 01945 (617) 631-4685.

DDJ



Artwork

Professional graphics capability on a personal computer for $500.00 (IBM color graphics card) is possible with **Artwork**, an interactive graphics-design program. With the use of an input tablet or mouse, a designer can draw, modify, edit, and store complex graphic elements. Artwork includes a library of type fonts that can be scaled, rotated, italicized, or condensed. The fonts are vector-defined, not bit-mapped, so they can be changed like any other image created by Artwork. The program also offers three-dimensional graphics manipulation capability. Images created by Artwork can be retouched by Artpaint and vice versa. The two programs are designed to complement each other. Live images from a video camera can be captured and manipulated by Artpaint. Address: West End Film Inc., 2121 Newport Place, NW, Washington, DC 20037 (202) 223-2938.

PC/AT Internal Expansion Kit

A hard disk and 1/4-inch tape drive internal upgrade kit for IBM's PC/AT converts the AT into a virtual mainframe. The kit uses IBM's hard disk controller. The Back Up and Restore Utility backs up 65 megabytes in 12 minutes. The system also allows user-configurable disk caching of up to 4 megabytes. The kit (which fits inside the AT's casing) is available in 40, 70, 140, and 280 megabytes. The internal 280 megabyte hard disk kit is priced at $15,850.00 from Emerald Systems Corporation, 4901 Morena Boulevard, San Diego, CA 92117 (619) 270-1994.

Index

16-Bit Software Toolbox (Column) 60, 136, 275, 414, 516, 611, 777, 882, 963
Accent Finder (Pascal) 341
Adding Primitive I/O Functions to muLISP 820
Algorithm, Minimax 491
Ashdown, I. 879, 896
Ashworth, R. 134, 135, 209

Balancing Act: The Ultimate Checkbook Balancing Program (BASIC) 512
Ballinger, C.K. 206, 209, 210
Barnhart, J. 652
BASIC *see also* MBASIC, NBASIC
  Dates 8
  BASIC Language Telecommunications Programming 123
  BASICFMT for TRS-80 (BASIC Formatter) 253
Bass, J.L. 930
Bergman, E.E. 506
Bibliography, Unix Internals 939
Binary Magic Numbers 202
BIOS Modifications, CP/M on the Commodore 64 364
Blum, R. 56, 89, 155, 223, 296, 444, 639, 818, 916
Bomberger, A. 826
Book Reviews *see* Reviews, Books
Brooks, L. 96
Broughton, R.S. 123
Burton, C.E. 157
Burton, R. 236
Buvel, R. 674

C *see also* Small-C
  C/Unix Programmer's Notebook (Column) 94, 282, 417, 618, 708, 783, 967
  GREP.C Parser 749
  A Guide to Resources for the C Programmer 862
  Optimizing Strings in 226
  RESORT (User Dictionary) 873
  Revised Program 6
Cadmus, R. 644
Calculations with Complex Numbers 734
Carroll, D.W. 65, 208
Carter, M. 820
Cashton, D. 132, 802
cc: A Driver for a Small-C Programming System 386
Checkbook Balancing Program 512
Chinese Forth *see* Forth
Clark, D.D. 409, 734, 976
Clark, R. 413
Coleman, R. 99
Commodore 64, CP/M on 364
Communications
  BASIC Language Telecommunications Programming 123
  Communications Protocols: Theory and Practice 96
  Micro to Mainframe 93
  Unix-to-Unix Network Utitilies 99
  Virtual Personal Computer Project 101
Compilers
  Forth Native-Code Cross Compiler for MC68000 674
  PL/C Programming Language 176
Computer Calisthenics (Column) 895, 980
Computing with Streams 663
Cortesi, D.E. 10, 29, 86, 153, 222, 294, 362, 441, 540, 547, 634, 722, 816, 914
Coy, K. 93
CP/M
  CP/M 2.2 Goes PUBlic 843
  CP/M Exchange (Column) 56, 89, 155, 223, 296, 444, 639, 725, 818, 916
  on the Commodore 64 364
CP/M Plus
  File Comparator for 547
  Resident System Extensions 455
Crayne, D. 68, 514
Crockett, D. 253
Cryptography 157

Daetwyler, D. 41
dBASE II
  Programming Techniques 375, 731
  Quirks in 353
DEC
  Emulate WordStar on TOPS-20 54
  System-10, Micro-Mainframe Connection 93
DeGroot, D. 66
Diskette Directory Listing (IBM PC) 41
Doherty, M. 406
Dovey, M. 176
Dr. Dobb's Clinic (Column) 10, 86, 153, 222, 294, 362, 441, 540, 634, 722, 816, 914
Duncan, R. 60, 136, 211, 275, 305, 414, 516, 611, 704, 777, 882, 963

Editorials 3, 83, 219, 360, 438, 528, 630, 718, 812, 908
Emulate WordStar on TOPS-20 54
Expert Systems and the Weather 231

Fast Fourier Transform (Forth) 652
Favitta, M. 69
Fig-Forth *see* Forth
File Browser Program (C) 946
File Comparator (CP/M Plus) 547
File Maintenance in Forth 644
First Chinese Forth: A Double-Headed Approach 379
Fisher, S. 54
Fleming, J. 101
Fletcher, G. 543
Formatter, BASIC for TRS-80 253
Forth
  and the Fast Fourier Transform 652
  Converting Fig-Forth to Forth-83 305
  File Maintenance in 644
  First Chinese Forth: A Double-Headed Approach 379
  Languages and Parentheses, Forth-like 506
  Native-Code Cross Compiler for the MC68000 674
  Standard Floating-Point Extension 704
  to PC-DOS Interface 29
Forth-83 *see* Forth
FVG Standard Floating-Point Extension 704

Gomez, A. 971
Greek and Math Symbols with WordStar 543
GREP.C: A Unix-Like, Generalized, Regular Expression Parser in C 749
Grigonis, R. 312
Grossman, A. 93
Guide to Resources for the C Programmer 862

Halcomb, J. 135
Head, G. 353, 375, 731
Hendrix, J. 322

Hinson, E.F. 267
Holub, A. 749
Huang, T. 379

I/O Functions (muLISP) 820
IBM PC
  Forth to PC-DOS Interface 29
  Sorted Diskette Directory Listing 41
  Window Package for 21
Infinite Key Encryption System 588
Interrupts, Program Monitor Package (CP/M) 826
Introduction to Parsing 956
Introduction to PL/C: Programming Language for Compilers 176
Irving, R. 209

Johnson, J.R. 513, 946

Kachigan, K.R. 972
Kaplon, M.F. 805, 900
Key Encryption System (CP/M) 558
Krantz, D.G. 873
Kronman, J. 131

Langevin, R.A. 130, 804
Languages and Parentheses: A Suggestion for Forth-Like Languages 506
Letters (Column) 4, 84, 152, 361, 439, 631, 719, 813, 909
Levy, S.G. 5
LISP *see* muLISP
Louis, G. 71
Lozes, R.L. 208, 898

MBASIC, Structured Preprocessor for 14
MC68000, Forth Native-Code Compiler for 674
McDermott, E. 226
McKay, D. 843
McKillip, M. 127
McLarty, H. 300
Micro to Mainframe Connection (Radio Shack Model III to DECsystem-10) 93
Minimax, A Simple Algorithm (BASIC, Polyforth, FML) 491
Mitchell, B. 843
Mitchell, E. 21
Modem, U.S. Robotics' S-100 127
Modula-2 for Pascal Programmers 200
Moran, J. 65, 208, 413
More dBASE II Programming Techniques 731
More on Binary Magic Numbers 202
muLISP, Primitive I/O Functions 820

NBASIC: A Structured Preprocessor for MBASIC 14
New Library for Small-C 322, 397

Odette, L.L. 663
Of Interest (Column) 78, 146, 215, 284, 355, 433, 521, 663, 625, 712, 806, 902, 981
Optimizing Strings in C 226
Ordman, E.B. 803
Osborne 1, Window Package for 21
Ouverson, M. 67

p - A Small C Preprocessor 464
PABX and the Personal Computer 120
Park, J. 231
Parsers
  GREP.C 749
  Introduction to Parsing 956
Parsons, R.G. 623, 880
Pascal Programmers, Modula-2 for 200
Payne, E. 322

PC-DOS, Forth Interface to 20
Petzold, C. 206
Piotrowski, W. 364
PL/C, Introduction to 176
Program Design Using Pseudocode 195
Program Monitor Package: Using Interrupts to Instrument Applications 826
Provenzano, T. 899
Pseudocode 195
Public Key Cryptography System (RSA) 157, 236

Radio Shack
  Model III, Micro-Mainframe Connection 93
  TRS-80 BASIC Formatter 253
Resident System Extensions: RSX Under CP/M Plus 455
RESORT (C) 873
Reviews, Book 65, 134, 208, 413, 513, 802, 896
Reviews, Software
  COMPAC 68
  FILEBASE 132
  Friday! 130
  McDisplay 69
  METAFILE 268
  microSUB:MATH 267
  MYCHESS 880
  NCI COHERENT 971
  Oubliette 133
  PC Small-C 623
  PC-Write 623
  Perfect Writer Perfect Speller 206
  SAL/80 131
  SMAL/80 71
  SuperScripsit 67
  SXR Plus 206
  Turbo Pascal 409, 971, 972, 976
  Windows for C 879
Revised C Program 6
Rogers, J. 939
Rohling, T. 120
Rothstein, J.B. 134, 513, 897
RSA: A Public Key Cryptography System 157, 236
Ruzinsky, S. 491

Schreiner, A.T. 386, 464
SCISTAR: Greek and Math Symbols with WordStar 543
Seymour, H.A. 956
Shammas, N. 14
Silvey, G.M. 455
Simple Calculations with Complex Numbers 734
Simple Minimax Algorithm 491
Simple Window Package (IBM PC, Osborne 1) 21
Sixth Generation Computers 312, 406
Skjellum, A. 143, 282, 417, 618, 708, 783, 967
Smal/80 73
Small-C *see also* C
  Concordance Generator 593
  cc: A Driver for a Small-C Programming System 386
  A New Library for Small-C 322, 397
  p — A Small C Preprocessor 465
Smith, D.W. 300
Software Designer (Column) 727, 894
Software Reviews *see* Reviews, Software
Solutions to Quirks in dBASE II 353
Sorted Diskette Directory Listing for the IBM PC 41
Staneff, J. 593
Stith, J.E. 512
Streams, Computing with 663
Suckow, H. 5
Sutherland, R.P. 806, 902, 981
Swaine, M. 360, 438, 528, 630, 718, 727, 812, 894, 908

Takara, K. 195
Telecommunications *see* Communications
Thersites, J. 558
Thomas, J. 543
Thomas, J.A. 558
TOPS-20, Emulate WordStar on 54
Tracy, M. 704
TRS-80 *see* Radio Shack

U.S. Robotics' S-100 Modem 127
Unix
  C/Unix Programmer's Notebook *see* C
  Device Drivers 930
  GREP.C, Parser in C 749
  Internals Bibliography 939
  Unix-to-Unix Network Utilities 99
  Varieties of 918

Varieties of Unix: An Introductory Guide to Microcomputer Unixes 918
Vasile, E. 341
Virtual Personal Computer 101

Walworth, A. 918
Ward, T.A. 862
What's the Diff? A File Comparator for CP/M Plus 547
Wiesenberg, M. 78, 146, 215, 284, 355, 433, 521, 625, 712, 895, 980
Wiggins, R. 3, 83, 151, 219, 358
Wiggins, W.J. 133
Wilson, D. 202
Windows, A Simple Package 21
Wordstar
  Emulation on TOPS-20 54
  SCISTAR: Greek and Math Symbols with 543

The Dr. Dobb's Series

Volume 1

Chronicles the advent of the microcomputer era in 1976. Always pertinent for bit crunching and byte saving, home-brew computer projects, and the technical history of home computing. Topics include: Tiny BASIC, the first words on CP/M, Speech Synthesis, Floating Point Routines, 6502 Disassembler for the Apple, and much more.

Volume 2

The small computer emerges as a powerful tool for the modern age in these issues from 1977. Topics include: Lawrence Livermore Lab's BASIC, Dr. Starkweather's PILOT, Using a Modem, String Handling Techniques, Ciphers, Turtle Graphics, micro utilities.

Volume 3

Explores the foundations of the mass market computer industry in 1978. Topics include: Programming in BASIC, PILOT, and Pascal, RAM Memory Testers, Apple utility programs, S-100 Bus Standard, STRUBAL: A Structured BASIC Compiler.

Volume 4

Heralds the widespread acceptance of the microcomputer in America. Innovative ideas and articles guide the reader through the age of discovery in 1979. Topics include: Selecting Business Software, Microcomputer Speech and Music, Information Networks, interfacing techniques.

Volume 5

Focuses on the technological promise of the modern microcomputer and the creative challenge facing prgrammers in 1980. Topics include: The revolutionary impact of CP/M, C programming and the UNIX operating system, survey of computer networks, software portability, introduction to FORTH, compiler writing.

Volume 6

The microcomputer enters the mainstream—these issues assess the revolutionary impact of microcomputers on the individual and on our society in 1981. Topics include: Computer Conferencing, The Power of FORTH, 8- and 16-Bit Technology, Rubik's Cube Simulator, An Adventure Game Development System.

Volume 7

Examines the potential of powerful 16-bit micros including the birth of the IBM PC in 1982. Topics include: In-depth coverage of FORTH, 68000, 8088 programming, C software tools, utilities for CP/M including a spelling checker, using bulletin boards, and more.

Volume 8

DDJ turns pro. Some of the most powerful, professional programmer's tools ever published in a magazine are in this volume. Jim Hendrix's Small C compiler. Ed Ream's RED screen editor. A microcomputer subset of the Defense Department's official programming language, Ada. C and Forth and 68000 software.

Volume 9

1984 was the year *DDJ* got mathematical with articles on the fast Fourier transform, a minimax algorithm, complex numbers, and two encryption systems. We explored the art of software design with articles on parsing, computing with streams, using pseudocode, and a new column on software design issues. There were articles on Chinese Forth, a new library for Small-C, an introduction to Modula-2, and a review of Turbo Pascal.

M&T BOOKS

ISBN 0-934375-08-9